Polsko-Japońska Wyższa Szkoła Technik Komputerowych

Katedra Multimediów

Multimedia - Programowanie Gier

Daniel Sadowski
Nr albumu s3305
Jarosław Socha
Nr albumu s3286

Development of an advanced real-time multimedia application in DirectX environment based on the game project "Nyx"

(Tworzenie zaawansowanej multimedialnej aplikacji czasu rzeczywistego w środowisku DirectX na przykładzie projektu gry komputerowej "Nyx")

Praca inżynierska
Napisana pod kierunkiem
Mgr inż. Krzysztofa Kalinowskiego

Warszawa, grudzień 2006

STRESZCZENIE

Niniejsza praca ma na celu przestawienie od podstaw procesu tworzenia gry komputerowej w środowisku DirectX na przykładzie projektu „Nyx". Omówione zostaną zarówno podstawy teoretyczne jak i praktyczne rozwiązania stosowane przy projektowaniu i implementacji multimedialnych aplikacji czasu rzeczywistego. Całe opracowanie zostało podzielone na cztery części. Część pierwsza definiuje pojęcia związane z tematem pracy. Część druga omawia teoretyczne podstawy projektu „Nyx", niezbędne do zrozumienia rozwiązań zastosowanych w fazie implementacji. Część trzecia omawia proces planowania oraz szczegóły implementacji projektu, natomiast ostatnia część niniejszej pracy zawiera jego podsumowanie oraz analizę.

TABLE OF CONTENTS

1. Introduction

The following thesis demonstrates the creation process of a real-time multimedia application in the DirectX environment. The authors present the theoretical background and the design and implementation process on the basis of a small computer game called "Nyx".

1.1. Document Structure

The following document has been divided into four separate sections. The goal of the first section (Introduction) is to introduce and explain the topic and the goals of this thesis. The second section (Theoretical Background) contains the theoretical basis of the project. The third section (Design & Implementation) contains the design and implementation details of the application, as well as descriptions and explanations of the exact approaches used by the authors based on the theoretical knowledge contained in section two. The fourth and final section (Conclusions) contains the overview of the final results and the authors' final conclusions.

1.2. The Goals

The main goal of the following thesis is to demonstrate the creation process of a real-time multimedia application using DirectX as the set of APIs[1] of choice. By presenting the essential theoretical knowledge and the design and implementation process of a small computer game called "Nyx" the authors demonstrate the complexity of such an undertaking.

The authors believe that creating a computer game is a science in itself because the amount of research and work necessary to complete such a task is enormous. It is not a trivial task in any way as game engines are one of the more complex projects anyone may attempt to complete. Professional engines are written by groups of highly skilled and experienced programmers, but the authors want to prove that one or two people are still capable of implementing a fully functional game within a reasonable amount of time.

It was not the goal of the authors to create a game containing many hours of gameplay[2] since such a task would require at least twice as much time as it took to complete the "Nyx"

[1] API – Application Programming Interface provides a set of services, functions and/or interfaces, which allow applications to communicate with external applications, libraries, operating systems or hardware.

[2] Gameplay refers to all experiences the player can have while interacting with the game. This includes all story content, sounds, music and also all challenges and tasks the player must complete.

project (11 months). Instead, the authors decided to concentrate on presenting as many techniques as possible in a short and small prototype of a game.

1.3. The Explanation of the Topic

The topic of the thesis deals with the development of an advanced real-time multimedia application in the DirectX environment. All theoretical and practical topics are presented using an application created by the authors as an example. However, in order to improve the clarity of the information contained in the whole document, it is important to provide brief explanations of certain terms contained in the topic of this thesis.

1.3.1. Real-Time Applications

A real-time application is special kind of a computer program. Instead of using an event-based architecture where operations are performed only after the user's input is detected, certain operations are performed in a loop, which updates the application regularly while additionally (but not necessarily) handling the user's input. As a result, it can change its state constantly, even without any user actions.

As hinted by their name real-time applications depend heavily on time constraints. They are expected to perform their operations within a short amount of time and as a result need to be very efficient and fast. Their performance is counted in the amount of operational cycles per second.

1.3.2. Multimedia Applications

Multimedia applications make use of many types of media content, such as texts, graphics, sounds, 3D visualizations, animations and videos simultaneously to perform their operations and generate the results on the screen. Such applications are usually used to inform or entertain the users and often provide them with high amounts of interactivity.

Examples of multimedia applications include: graphics and 3D modelling applications, various presentations, simulation programs, video assembly applications and also computer games.

1.3.3. Computer Games

Computer games are a very specific mix of the functionality of both real-time and multimedia applications. Games, just like multimedia applications, use many different types of media to generate their content in real-time, while providing the user with high amounts of interactivity. They allow the users to control the game's characters and sometimes even the

environments. But not all games are made for pure entertainment. Some can be educational and are used to help the users learn different things such as mathematics or foreign languages, and some are even used to train professionals such as doctors or pilots. However, the "Nyx" project was developed to be an example of a game, the goal of which is to entertain the user.

1.3.4. DirectX Environment

DirectX is a collection of APIs which are used during the development of many applications that render 3D images in real-time. It provides interfaces which handle communication with graphics cards, sound cards and also input devices. Because of the popularity of DirectX and its wide use in commercial computer games it was chosen to be the basis of the "Nyx" project.

2. Theoretical Background

The following section presents the theoretical knowledge, which is essential to fully understand the complexity of the "Nyx" project. All of the topics were researched by the authors and built upon to create a fully functional computer game prototype.

The chapters contained in this section present explanations of different technologies and terminology and describe the general methodologies behind some of the more complex techniques used in the project. They do not, however, contain the exact implementation approaches used by the authors as the goal of this section is only to create a basic understanding of all the different topics.

The topics presented in the following section include:

- Computer games as a newest form of entertainment and storytelling
- Use of graphical accelerators and DirectX in modern game productions
- 3-dimensional object modelling using newest tools and technologies
- Real-time rendering basics
- Shader technology and its uses in real-time rendering
- Dynamic lighting and shadowing techniques
- Post-processing and particle effects as a way to increase the quality of real-time rendered scenes
- Sound in games
- Collisions detection and interaction with the game's world
- Real-time rendering optimizations using space partitioning
- Artificial intelligence as a tool for crafting involving gameplay and enemies

2.1. Computer Games

As mentioned at the beginning of the thesis, computer games are programs which provide the users with high amounts of interactivity. By allowing the player to explore and interact with the virtual worlds, realities and characters they create a very unique form of entertainment and storytelling.

2.1.1. Short Description of Computer Games

Since their inception in the early years of the second half of the twentieth century computer games have evolved and changed very dramatically, finally becoming a cultural phenomenon and even a form of art. Children, adults, men and women – all of them are able to find something unique in computer games that makes them feel involved and interested in this, still young, form of entertainment.

Computer games offer a lot more than traditional games. Their biggest values are interactivity and storytelling. By creating an illusion of freedom of choice, the players are able to explore the game in any way they want and create their own style of solving different tasks and problems. All that is usually tightly wrapped in a storyline, which is the driving point of everything that happens in the game. Instead of observing the adventures of different characters, the players can become them and have their own adventure.

There are several different kinds of games and just like movies or books they can be divided into several genres, the most popular of which include:

- **cRPG or Computer Role-Playing Games**, which are usually based on fantasy stories and worlds, allow the players to explore the game's locations freely and in a nonlinear fashion. Players are able to choose different classes and races of their characters and develop them in any way they see fit. cRPG games are expected to contain tens of hours of playable content in forms of different quests and tasks, out of which only a small percentage relates to the game's main storyline. While many people enjoy such freedom and amount of things to do, others criticize cRPG games for being very schematic and containing tens of hours of very similar content.
- **RTS or Real-Time Strategy Games**, which are basically war-games, the key components of which are resource gathering, base building, technology development and control over the units (soldiers, vehicles, etc.).
- **FPS or First-Person Shooters**, which characterize games that use shooting and

fighting as primary gameplay elements and show the game's world from the controlled character's perspective.

- **Adventure Games**, which are characterized by complex, mysterious and often also funny character-driven storylines. There are many types of adventure games ranging from dialogue and inventory-based point and click games to puzzle-based first person perspective games.
- **Action Games**, which are characterized by very fast gameplay that includes a lot of fighting as well as navigating though the environment by climbing, jumping, etc. Always included in such games are simpler or harder environmental puzzles. This genre evolved from an older and still popular games called **Platformers**, which are two-dimensional games similar in concept to action games.
- **Action-Adventure Games**, which are a mix of concepts of action and adventure games.
- **Survival Horror**, which is a very specific combination of action and adventure games. It can feature many of the different elements of both genres but always have one thing in common. The gameplay is centered around settings and storylines that are supposed to scare the players and raise their level of tension. Often featured in such games are different kinds of demons and monsters, which usually try to kill the player's character at every occasion.
- **Sports Games**, which feature gameplay centered around a specific sport or activity such as playing football, basketball or racing.
- **Puzzle Games**, which contain gameplay centered around problem solving.
- **Massive Multiplayer Online Games**, which are slowly becoming more and more popular and use elements of RPG, FPS or Adventure games and expand on them to allow simultaneous play of many hundred of thousands of players. Such games are usually very community-oriented and provide a lot of challenges that should be solved together by groups of players.

2.1.2. Short History of Computer Games

The history of computer games spans only half of the century but it describes one of the fastest growing forms of entertainment ever to be developed. Describing it completely with all the details would probably require months of research therefore the history described here will only concentrate on some of the developments and games created in the last 50 years.

Figure 1: The first graphical computer game ever created.
Image Source: http://en.wikipedia.org/wiki/OXO (as of December 2006)

The first known graphical computer game was called **OXO** *(see Figure 1)* and was developed by A.S. Douglas in 1952 on an EDSAC computer. It was a game of tic-tac-toe, which allowed the user to play against the computer. The next computer game was developed by William Higginbotham in 1958 in order to cure the boredom of the visitors of the nuclear power plant he worked in. The game was called **"Tennis for Two"** *(see Figure 2)* and was displayed on an oscilloscope with all calculations being performed on a small analog computer. The game itself was very simple. It showed a simplified tennis court from the side and a small ball, which had to be played over the net.

Figure 2: The next three computer games.
Image Sources: http://en.wikipedia.org/wiki/Tennis_for_Two (as of December 2006)
http://en.wikipedia.org/wiki/Spacewar (as of December 2006)
http://en.wikipedia.org/wiki/Pong (as of December 2006)

Four years later, Stephen Russell from the Massachusetts Institute of Technology**,** developed another game called **"Spacewar!"** *(see Figure 2)* on a DEC Digital PDP-1 computer. The game allowed two players to control two different spaceships caught by the gravity forces of a star. Those spaceships had a limited number of missiles with which they could shoot each other. The game allowed the players to rotate the ships clockwise and

counterclockwise, thrust, fire and also jump into hyperspace, which basically meant that the ship would could disappear and reappear in a completely random location.

While **"Tennis for Two"** and **"Spacewar!"** were revolutionary creations, the most famous of the first games was **"Pong"** *(see Figure 2)*, which was released by Magnavox using Ralph Baer's designs in 1972. The game was similar to **"Tennis for Two**" in such a way that it also simulated a simplified "tennis" court with the difference being that it was showed from the top. Two players could play the game by controlling two paddles that the ball could bounce off. **"Pong"** was later re-developed by a now legendary company called Atari, which later developed many other games and also created a series of Atari home game consoles. After a while came a time when many games did something new for the first time. In 1979 Namco Company released **"Galaxian"**, which was the first color game and in 1980 for the very first time a speech system was used in **"Stratvox"** *(see Figure 3)*.

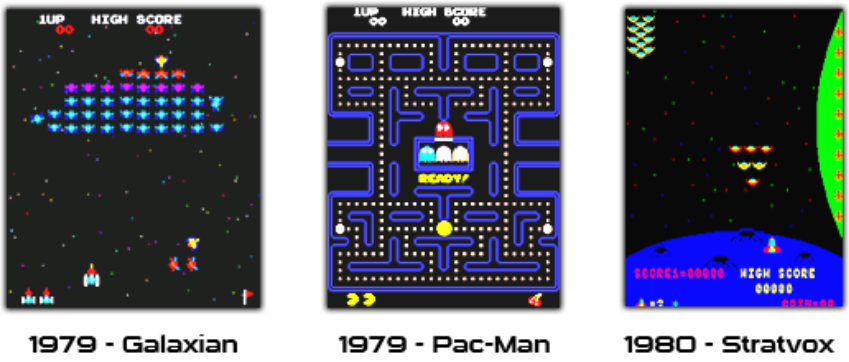

Figure 3: Screenshots from Galaxian, Pac-Man and Stratvox
Image Sources: http://en.wikipedia.org/wiki/Galaxian (as of December 2006)
http://en.wikipedia.org/wiki/Pacman (as of December 2006)
Authors' Personal Screenshot Library

During 1980's the game industry went through a number of changes and grew at an incredible speed but just before that happened, in 1979, a now legendary game called **"Pac Man"** was released.

The availability of personal computers and console systems such as Atari, Amiga, Commodore, ZX Spectrum and Nintendo, resulted in an extreme increase in game production. Many great and now legendary games such as "**Tetris"** *(see Figure 4)* and **"Super Mario Bros"** were released during those years. But great games were also being created and developed in Poland, with **"Robbo"**, **"Misja"** *(see Figure 4)*, **"Fred"** or **"Hans Kloss"** being one of the best and most popular.

Figure 4: Screenshots from Tetris, Robbo and Misja
Image Sources: Authors' Personal Screenshot Library

In 1990's the world of games changed once again. With PC computers becoming more and more popular and the releases of many new gaming consoles, games for older computers such as Amiga, Atari or Commodore no longer were being created. More powerful machines allowed more complex games, as well as new genres to be developed. Out of many games released in the early 1990s, worth mentioning are "**The 7th Guest"**, "**Myst"**, "**Alone in the Dark"** and "**Wolfenstein 3D"** *(see Figure 5)*, all of which popularized many new genres and styles of gameplay.

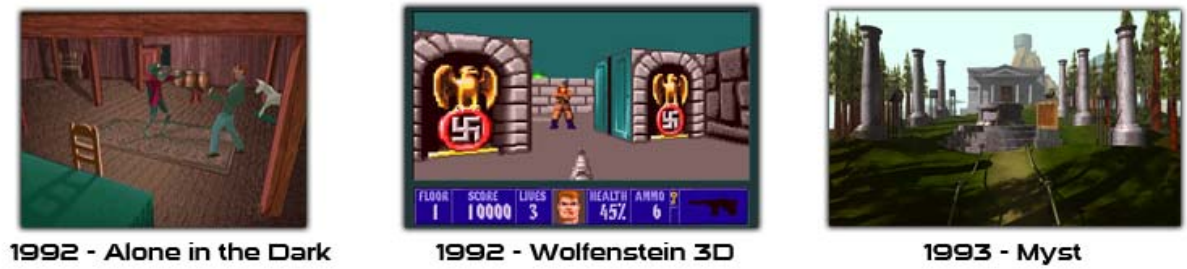

Figure 5: Screenshots from Alone in the Dark, Wolfenstein 3D and Myst
Image Source: Authors' Personal Screenshot Library

"The 7th Guest" (1992) and **"Myst"** (1993) were one of the first games to make use of the CD-ROM drives on PC computers. They popularized the puzzle adventure genre to such an extent that many companies, such as Cryo and Polish Detalion, were created with the goal to almost solely produce first person adventures. However, **"Myst"** and **"The 7th Guest"** were not the very first of their genre and were predated by first perspective adventure games, such as Polish **"AD2044"** and **"Klątwa"** developed for Atari computers.

"Alone in the Dark" (1993) originated a genre known as Survival Horror, by introducing a unique style of gameplay, where the character moved around 3D-like locations and was seen in the third perspective from many static cameras located in all of the game's locations. The game became so popular that not only two sequels were released in 1994 and 1995, but also

many similar games (**"Resident Evil"**, **"Silent Hill"**) were created by other companies.

Last, but not least, **"Wolfenstein 3D"** (1992) was responsible for popularizing the first person shooter genre, which is perhaps the most popular genre nowadays.

New games and new styles of gameplay hinted at the future of games as a form of entertainment and storytelling. It was not until mid-1990's, however, that the future of games became clear. With the introduction of 3D graphics accelerators a new era of gaming began. From that moment on the quality and realism of games would increase incredibly with each next year and each next revolutionary game release. New consoles would be released every couple of years and new PC hardware would be released every year increasing the performance almost by a 100% every time. Increased availability of the internet and fast connection speeds would popularize online gaming so much that almost half of the new releases would contain an online multiplayer mode.

Since the creation of the first game, the world of gaming has changed dramatically. Nowadays, games are rarely created by a single person. Large commercial titles usually require teams of even 50 people, all with different skills, to bring the game from a concept to a fully working application.

The year 2006 marks the introduction of the seventh generation consoles, such as Playstation 3, Wii and Xbox 360, which are capable of generating ultra-realistic graphics in real-time. Additionally first DirectX 10 compatible graphics accelerators for PC computers were released in that year. Games are becoming more and more realistic and the large development companies are controlling most of the market with franchise titles. Unique games are now created mostly by independent developers and smaller development companies. Small games like those created in 1980s are now referred to Casual Games and are mostly from the old arcade[1] or puzzle genres.

2.2. Graphics Accelerators

One of the most important inventions in the game industry is a graphics accelerator, which truly revolutionized the way the games look and are created. It is a type of a video adapter that contains a processor specialized for computing 3D graphical transformations.

Modern graphics accelerators allow hardware acceleration of vertex transformations, texturing and lighting. Because of that such operations no longer need to be performed by the

[1]Arcade games are mostly short and simple, easy to learn games, which provide the player with intuitive, quick and fun gameplay.

software on the CPU[1]. This allows programmers to use the extra processor time freed by the accelerator to perform more complex software computations in other areas of the application such as artificial intelligence.

2.2.1. Short History of Graphics Accelerators

The era of 3D game acceleration began with the release of the 3Dfx Voodoo card in 1996. This card has made an enormous impact on the game industry as it changed the way all future games would look and be created. The Voodoo cards were unique in their design as they were targeted for 3D games. However, 3Dfx was not alone in the market and soon other companies such as Intel, ATI, NVidia, Matrox, SiS and Rendition began releasing their 3D graphics accelerators.

After 3Dfx's success, ATI released a Rage II graphics accelerator, which failed to outperform the widely popular Voodoo. Soon, however, cards like Rage Pro or NVidia's Riva 128 became its worthy opponents.

Voodoo 2, released in 1998, solidified 3Dfx's position on the market. The new card included a second texturing unit and a faster processor that effectively doubled the card's performance. Additionally, it became possible to connect two graphics cards together. Very soon other companies like NVIDIA, ATI and MATROX introduced even more powerful cards including: NVIDIA Riva TNT, ATI Rage 128 and MATROX G200 and then Riva TNT 2 and ATI Rage 128 pro.

Eventually, the revolutionary 3Dfx became defunct and all of its assets were acquired by NVidia, which along with ATI eventually gained control over the whole graphics accelerator market. In 1999 NVidia released its first GeForce card, while ATI released the first Radeon a year later. Both card series are being constantly developed and updated with new models released every year. Nowadays graphics accelerators are capable of performing operations on enormously complex scenes with many different effects and textures being applied.

2.3. DirectX API

DirectX is a collection of APIs for handling tasks related to game programming. It is a free API that is most commonly used in development of computer games for Windows and XBox.

---

[1] **CPU**, or **Central Processing Unit**, is an integrated circuit present in all modern computers, which interprets computer program instructions and processes data.

2.3.1. Contents of DirectX API

DirectX (9.0c version used by the authors) consists of the following APIs:

- **Direct3D**, which provides the programmers with access to the graphics accelerator device. It also includes a set of functions related to the rendering process and a set of mathematical functions, which perform common operations on matrices and vertices. Direct3D also can emulate vertex operations on the CPU if no hardware processing is possible on the installer graphics card.
- **DirectSound**, the functions of which include support for ambient and 3D sounds as well as volume control and audio mixing.
- **DirectInput**, which supports input from the user by means of input devices such as keyboard, mouse and joystick. It provides functionality of action mapping which allows for action assignment to the buttons and axis on the input device.

Older, and now mostly deprecated, parts of DirectX include:

- **DirectDraw**, which was responsible for 2D graphics rendering. All of DirectDraw functions can be performed using Direct3D.
- **DirectShow**, which was used to perform audio and video streaming. It allowed playback of the multimedia files and also image and sound sampling from external sources like digital cameras.
- **DirectPlay**, which performed tasks related to networking.
- **DirectMusic**, which was a high level API related to playback of sound and music.

2.3.2. Short History of DirectX

DirectX runtime was originally distributed only by game producers along with their games to ensure that a correct version of DirectX was present in the player's system. Nowadays, however, it is by default also included in the Windows operating systems.

The history of DirectX begins in 1994 when Microsoft was about to release Windows 95. The success of the system was very much based on what programs customers would be able to run on it. Several of Microsoft's employees, such as Craig Eisler, Alex St. John, and Eric Engstrom, where concerned that programmers would still consider DOS, which was Microsoft's previous operating system, as a better platform for game programming. That would mean that few games would be developed for Windows 95 and the system would not be as much of a success. Therefore, Microsoft needed to develop an API targeted solely at game programmers, which would provide functionality that would simplify and speed up the

development of games. Eisler, St. John, and Engstrom eventually developed such an API, which premiered in 1995 under the name of Windows Games SDK. Later, it would be known as DirectX.

After the release of DirectX a battle began between Microsoft and the developers of OpenGL[1], which was another graphical API. This competition resulted in the very fast development of new versions of DirectX. In one year after the first release of DirectX two more versions were released: DirectX 2 and DirectX 3.

Along with DirectX 3 a new API was introduced called Direct3D. It was based on the technology developed by Rendermorphic, a company acquired by Microsoft, and provided functionality dealing with the rendering process of real-time 3D content.

In the following years new version would be released regularly, each providing programmers with new functionality and interfaces. Eventually, DirectX became an industry standard and is now widely used in game development.

The were several reasons for the success of DirectX. Just like OpenGL, it made game development a much simpler and quicker process by providing interfaces that supported all graphics accelerators. But the main difference between the two APIs was that Microsoft was constantly updating and supporting theirs, while maintaining a very detailed documentation of the API. OpenGL, on the other hand, remained open-source and did not change as much nor as often as DirectX did.

2.4. Mesh Technology

Every game requires content. Without it, there would be no gameplay and without gameplay there would be no game. When 2D games were still very popular, game content was either hand-drawn or prepared using a varied range of vector or raster graphical applications. With time, however, game developers moved their games and ideas from the 2D world to the 3D world. This changed things dramatically as programmers and artists were

[1] **OpenGL**, which stands for **Open Graphics Library**, is standard specification of a cross-language, cross-platform API designed for applications, which produce 2D or 3D real-time graphics. It was developed by Silicon Graphics in the early 1990's and later updated to version 2.0 by 3DLabs in 2004. Because in its most basic form OpenGL is just a specification, the functions it describes have to be implemented by hardware manufacturers. Additionally, it allows individual vendors as well as independent developers to add more functionality to the standard if required. OpenGL is commonly used in scientific and information visualizations as well as CAD (Computer Aided Design, which is used by engineers). Moreover, it is also used in computer game development, but not as widely as DirectX.

forced to adjust to the new environment and come up with new ways and techniques to create their games. While programmers eventually moved on to 3D APIs like DirectX or OpenGL, artists were forced to create the content using 3D modelling tools.

2.4.1. Polygonal Meshes

There are several ways to represent a 3D object mathematically. One way requires the usage of NURBS[1], which are basically curved surfaces calculated from a series of control points using, for example, Bézier curves[2]. However, in game programming, content creation and also real-time generated graphics there is only one widely used method of 3D object representation. It is called polygonal meshes.

A polygonal mesh is basically a collection of interconnected vertices (or points in 3D space), which define the shape of the 3D object. Two connected vertices form an edge, while three or more neighboring edges create the so-called polygons *(see Figure 6)*. Hence the 3D object is eventually represented by a set of geometric shapes (faces) defining its boundaries.

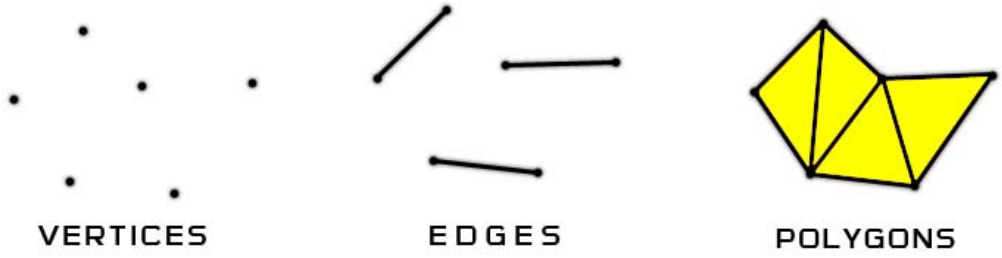

Figure 6: The elements of the mesh - from left to right: vertices, edges and polygons (or faces)

In games, the object is always represented as a set of triangles. The reason is that a triangle precisely defines the surface on which it lies and there will always be a plane that can intersect all three vertices of a triangle, which may not be the case with higher polygons (4 or more vertices per face). It also allows for a simple way to interpolate the values of positions, colors, normals[3] or texture coordinates of the vertices over the whole surface of the triangle, which is especially useful in lighting and texturing calculations.

---

[1] **NURBS**, or **non-uniform, rational B-spline**, is mathematical model used in computer graphics for the representation and creation of curves and surfaces.

[2] **Bézier curves** are parametric curves developed by a French mathematician Pierre Étienne Bézier. Along with **Bézier surfaces** (which are a generalization of Bézier curves), they are used in computer generated graphics and allow generation of very smooth surfaces. The curves are calculated from several control points (commonly 3 or 4 in computer applications).

[3] A normal of a flat surface is a vector that's perpendicular to that surface. It indicates a direction the surface is facing. Normal vectors are used in lighting computations.

2.4.2. The Tools

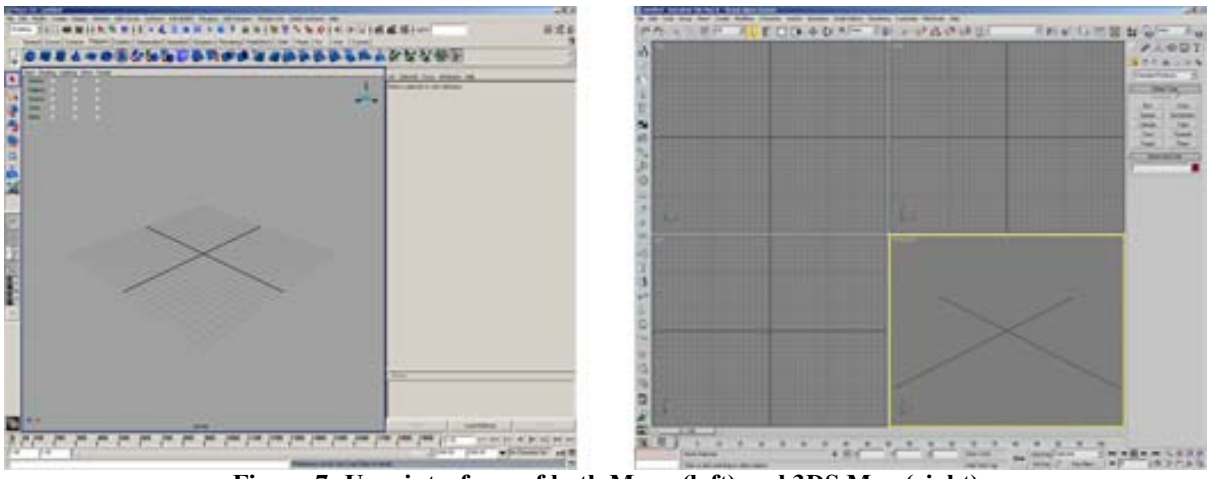

Figure 7: User interfaces of both Maya (left) and 3DS Max (right)

There are many applications that help artists create 3D objects, but the two currently leading and most powerful of them are 3D Studio Max[1] and Maya[2] *(see Figure 7)*. Both provide the most complete tools for 3D object creation and are widely used in television, movie and game industries. They allow artists to model and texture the objects and also animate them if necessary. Unfortunately, both 3DS Max and Maya are very expensive and as a result they are mainly used in heavily-sponsored professional projects. Since not everyone is able to afford such expensive modelling applications another way had to be found for independent game developers[3] and artists to create their games. Fans of the so-called "indie" game development have been seeking ways to create cheap content for their games and it finally became possible with the introduction of Blender[4].

Blender provides most of the tools 3D Max and Maya provide, while being free and open-sourced. This opens the doors for many changes and possibilities. Not only Blender might soon become a real tool-of-the-trade securing its rightful place next to Maya and 3DS Max but it might also become a more powerful tool than any of those two currently leading programs. Even right now Blender is being used in more and more independent games and it

[1] http://www.autodesk.com/3dsmax

[2] http://www.autodesk.com/maya

[3] Independent Game Developers are game developers that are not controlled by the publishers as they have no written contracts with any of them. They are usually small groups of people that create their games over the internet and later self-publish their works. By retaining all creative control over their project, "indie" developers usually push the boundaries of gameplay design creating original and innovative gameplay.

[4] http://blender.org/

was even used in the preproduction stages of Spider-Man 2 the movie [Blender]. It is just a matter of time before professional game developers will start using it in their productions, if only to lower development and tool costs.

2.4.3. Low-poly modelling

Today's hardware is able to handle scenes which contain millions of polygons but even if the Graphical Processing Units[1] are able to handle such detailed scenes there are still some overheads generated by the GPU itself, 3D APIs (DirectX or OpenGL) and the engine that limit the number of polygons that can be handled. 3D games need to be interactive and the images visible on the screen should move smoothly. To achieve this, the GPU has to render the scene at least 25 times per second and for that reason it is important to keep the number of polygons as low as possible, while still keeping the level of detail that is desired. This creates many problems for the programmers and artists. Programmers need to be careful with applying various effects on the rendered images (especially where complex lighting is concerned) and limit the amount of information sent to the GPU to a minimum. Artists, on the other hand, need to try to create as simple meshes as possible while modelling the game's objects.

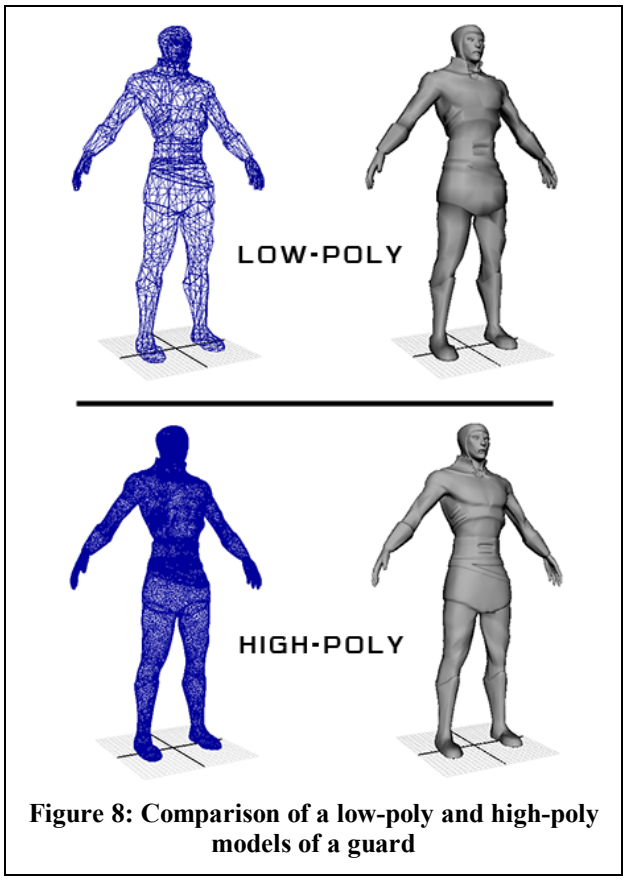

Figure 8: Comparison of a low-poly and high-poly models of a guard

Modelling for games is usually referred to as low-poly modelling *(see Figure 8)*. The term, however, is very flexible, as what is referred to today as low-poly would be consider very much high-poly few years ago. What it means is that artists can not add too much detail to the scene using polygons as they would be able to do in movie production. This is where the limitations and disadvantages of the polygonal meshes come. For example, to achieve

[1] **Graphical Processing Unit** is a dedicated rendering device for modern computers and consoles. Because of its specialized design, it can manipulate and display computer generated graphics more efficiently than a general purpose CPU.

curved surfaces it is necessary to approximate the curves using a set of triangles. Similarly, achieving very detailed objects in a scene requires artists to add more triangles to the objects' geometry. The more triangles are used the more curved and detailed the surfaces become. However, such method comes with a huge price. More triangles require more operations while rendering the scene and that, of course, requires more time and resources. While rendering time may not be as important in the movie industry or pre-rendered footage it becomes a problem when the image needs to be generated in real-time. That is why many details are added to the scene at render time using textures and shader effects and curves are simulated by a lower number of polygons. Nevertheless, both methods are not perfect and as a result the inaccuracies remain visible to some extent. There are, however, techniques to minimize the inaccuracies, some of which will be explained in the later chapters.

2.4.4. Skinning and Animations

One of the most important features of 3D creation tools like Maya and 3DS Max is the ability to apply skeletons to character models. By applying skeletons to the polygonal meshes artists are able to animate the objects more easily than if they had to animate each and every vertex by hand. This technique is called skinning *(see Figure 9)*.

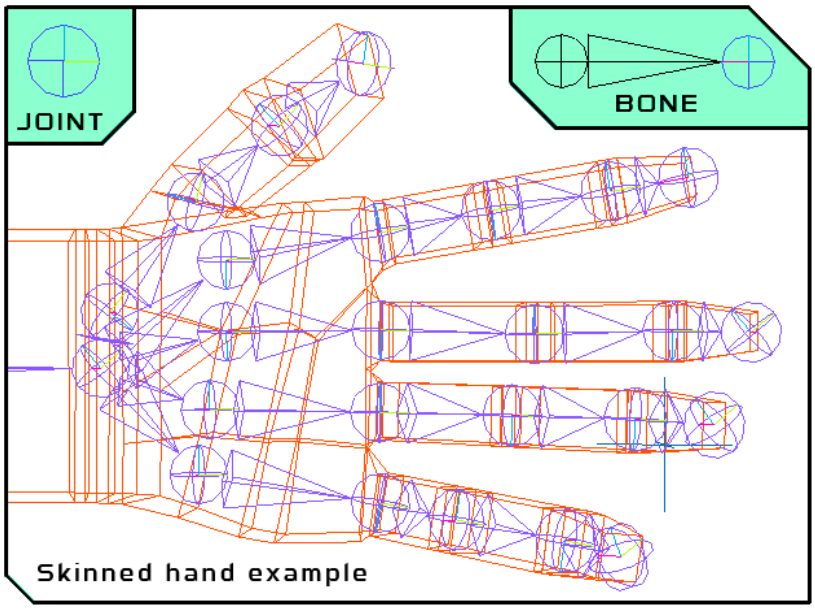

Figure 9: Skinned hand example.
It is visible that bones and joints of the hand are placed in similar places as in their real-life equivalent.

Skeletons in 3D models work very similarly to those of a human body. They consist of connected joints and bones and they can be used as a way to deform the mesh they are attached to. Just like in real life, a skeleton is surrounded by a skin, which in the case of a 3D

model is a polygonal mesh. Since each bone has a parent bone the whole skeleton is eventually grouped into a hierarchy of bones. Each bone influences a part of the mesh and other bones that are parented to it by changing their position, rotation and shape depending on the current position and rotation of the bone. For example, an upper arm bone would influence the lower arm, hand and finger bones – just like in real life.

The sets of vertices that bones influence are called "null nodes" (or "dummy nodes" or "grouping nodes") and are located in the positions of their joints. The distance between two child-parent nodes defines the length of the bone.

Different 3D programs treat joint and bones differently. For example Maya visualizes the bones between two joints while 3D Max visualizes only bones basing on their starting locations, directions and lengths.

2.4.5. Bones vs. Bipeds

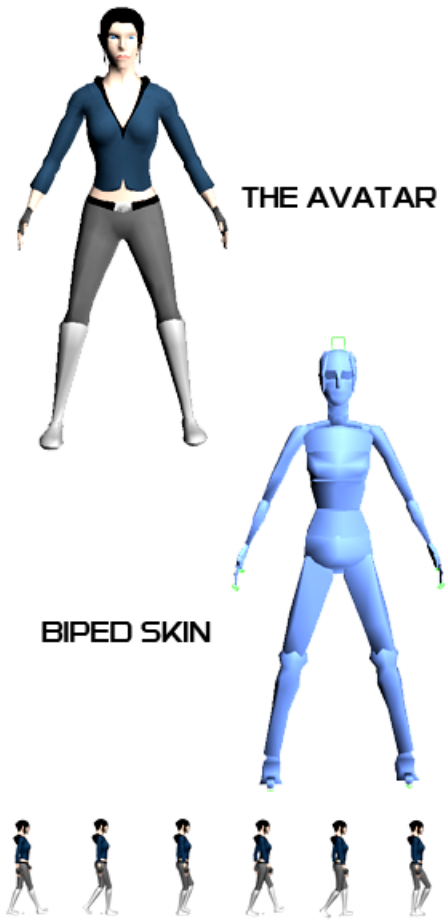

Figure 10: The biped skin of the game's playable character

Bipeds *(see Figure 10)*, which were first introduced in Autodesk's Character Studio (at first as a plug-in and later as part of 3DS Max 7), became a new approach to skinning character models. While working very similarly to bones, bipeds are based on a slightly different ideology. Instead of having bones to control the character, a kind of a ragdoll[1] system composed out of symbolic meshes is used. Like with bones, each part of the doll is hierarchically connected to another and influences a chosen set of vertices. The differences between both systems, however, come from the way they are set up. While bones are very symbolic, bipeds can be stretched to more visibly overlap parts of the skinned mesh. Setting up vertex influences is also simpler and more automatic than with bones.

There are many arguments for and against bipeds. Some artists argue that the biped skinning system is not professional enough and it is not possible to

[1] **Rag doll physics** - a program allowing character models to react with realistic body and skeletal physics.

control the characters as perfectly as with bones, while others say it is faster, easier and at the same time as flexible as the bone system. Some even foresee bipeds completely overtaking the standard bone systems looking at how Autodesk is building an increasingly stable and feature-rich biped and character animation system for 3DS Max with each new release. The truth is, however, that both methods give very good results and both are widely used in commercial games after, of course, some minor or major modifications to fit the game engine's animation system.

2.5. Real-Time Rendering Basics

Real-time rendering is a complex procedure. It requires many operations being performed on both CPU and GPU as well as many different modules communicating with each other. One of the ways to simplify the rendering process is to use a 3D APIs like DirectX. Still there are many concepts a programmer must understand in order to be able to create any real-time rendering applications [Dempski02].

2.5.1. Coordinate Systems

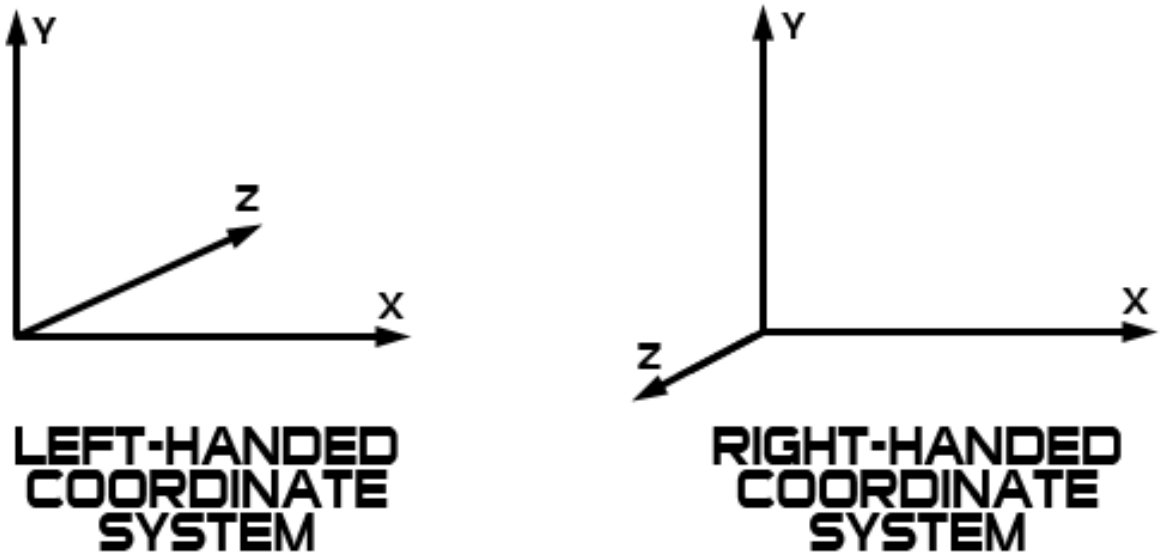

Figure 11: Left-handed coordinate system vs. right-handed coordinate system

Mathematical concepts are used extensively during real-time rendering, therefore understanding them is very important. The most basic of the used concepts is a **coordinate system** *(see Figure 11)*. In most simple terms an n-dimensional coordinate system allows to assign a set of numbers to any point in such a space. The origin is a middle-point of the system and has zeros assigned as its coordinates. For 3D space representation two different coordinate systems can be used: a so-called left-hand coordinate system or a right-hand

coordinate system. In both systems the X axis points to the right and Y axis points up. The Z axis, however, can point in two different directions as demonstrated on the picture below.

The next two essential concepts are **vectors** and **matrices**. Direct3D API provides methods capable of performing many calculations on them, as they are extensively used during the rendering process.

2.5.2. Vectors

An **n-dimensional vector** is a set of numbers describing a position in an n-dimensional coordinate system. In real-time applications 2-dimensional, 3-dimensional and 4-dimensional vectors are used. Some of the common uses of vectors include: description of the object's position, the amount of movement or a direction. Direct3D supports many vector operations such as length computations, dot and cross products as well as standard vector arithmetics.

Vector length is computed using the following equation:

$$\|a\| = \sqrt{a_1^2 + a_2^2 + \ldots + a_n^2}$$

What is worth mentioning is that a vector with length equal to 1 (called a unit vector) can be used as a directional vector. Such vectors are used to describe for example, light ray directions or movement directions.

Sometimes after performing several operations on vectors it is necessary to bring the vector's length back to 1. Such operation is called **normalization** and is performed by dividing a vector by its length.

The **dot product** can be calculated in two different ways:

$$a \bullet b = \langle a_1, a_2, \ldots, a_n \rangle \bullet \langle b_1, b_2, \ldots, b_n \rangle = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n$$

or:

$$a \bullet b = \|a\| \|b\| \cos\alpha$$

Both formulas can be combined together to calculate the angle between two vectors:

$$\alpha = arccos\left[\frac{a_1b_1+a_2b_2+...+a_nb_n}{\|a\|\|b\|}\right]$$

If a dot product between two unit vectors is calculated the resulting value is a cosine of the angle between the two vectors. Such operation is a very useful tool as it is commonly used during lighting computations and even artificial intelligence. For example thanks to the dot product it becomes easy to detect whether the an enemy is facing the player or whether the light rays affect any given object surface.

The **cross product** is yet another commonly used vector operation. For example it can be used to calculate normals of the polygons. It is calculated using the following formula:

$$a \times b = n\|a\|\|b\|\sin\alpha$$

The two vectors (***a*** and ***b***) describe a plane and the result is a vector that is perpendicular to that plane. There is of course another way to calculate the cross product. For a 3-dimensional vector the formula looks as follows:

$$a \times b = n = \langle n_1, n_2, n_3 \rangle$$

$$n_1 = a_2b_3 - a_3b_2$$
$$n_2 = a_3b_1 - a_1b_3$$
$$n_3 = a_1b_2 - a_2b_1$$

It is worth mentioning that while vector ***n*** is perpendicular to the plane created by vectors ***a*** and ***b***, it is not necessary a normalized vector.

The order of vectors in a cross product formula is important as changing the order changes the sign of the result:

$$a \times b = -(b \times a)$$

2.5.3. Matrices

In real-time rendering, **matrices** are used to perform different kinds of calculations on vectors and other matrices. If an object needs to be moved, scaled or rotated or a directional

vector needs to be rotated the matrices are the tools to do it. In simplest of terms a matrix is an NxM table containing numerical values. Direct3D makes use of only 4x4 matrices as all positions, rotations and scales are always represented as 3-dimensional or 4-dimensional vectors. In game programming matrices are closely related to terms such as **object space**, **world space**, **view space** and **screen space**. The first three spaces are all 3-dimensional, while screen space is 2-dimensional.

2.5.3.1. Object Space

Object space is a coordinate system which is local to the object. Meshes are always modelled in object space with their point of origin being usually in the middle of the model. Because every object has its own individual coordinate system it can easily be copied and transformed independently of other instances of the same object.

2.5.3.2. World Space

World space is the game's world. In this space all objects are placed and all movements, physics and collisions are calculated. All objects need to be transformed from object space to world space using a **world matrix**, which is basically a composition of three multiplied matrices: scale matrix, rotation matrix and translation matrix.

The **scale matrix** is used for scaling the object. It is constructed as follows:

$$\begin{bmatrix} \boldsymbol{a} & 0 & 0 & 0 \\ 0 & \boldsymbol{b} & 0 & 0 \\ 0 & 0 & \boldsymbol{c} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where: ***a***, ***b*** and ***c*** are the scale values in width, height and depth respectively.

The positions of all vertices are scaled using such a matrix, for example a vertex with position 2, 4, -1 could be scaled as follows:

$$\begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ -2 \\ 1 \end{bmatrix}$$

The **rotation matrix** is used for rotating objects. The full rotation matrix is the result of multiplication of three one-axis rotation matrices in the following order:

Rotation along the Y axis (yaw rotation matrix):

$$\begin{bmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation along the X axis (pitch rotation matrix):

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation along the Z axis (roll rotation matrix):

$$\begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The **translation matrix** transforms object's original position to a specified position in world space:

$$\begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where: ***a***, ***b*** and ***c*** are the translation values in 3-dimensional space.

The positions of all vertices are translated using such a matrix, for example a vertex with position 2, 4, -1 could be translated as follows:

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 1 & 4 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ 3 \\ 1 \end{bmatrix}$$

The order of multiplication of the scale, rotation and translation matrices is very important and to transform an object "correctly" it should first be scaled, then rotated and then translated. Of course, depending on what result is required the order of the multiplication can be changed. For example, if an object is first translated and then rotated then the object will be in a sort-of orbit around the point of origin of the world space coordinate system.

2.5.3.3. View Space

The **view space** (also referred to as "eye space" or "camera space") is a coordinate system that is relative to the camera. Objects are transformed from the object space to view space using a view matrix. To construct such a matrix three vectors needed. The **"eye"** vector, which describes the position of the camera, the **"look at"** vector, which describes the point the camera is looking at and the **"up"** vector, which is perpendicular to the "look at" vector and points to camera's top. The following is the formula for creating the view matrix using the left-handed coordinate system:

$$\begin{bmatrix} xaxis.x & yaxis.x & zaxis.x & 0 \\ xaxis.y & yaxis.y & zaxis.y & 0 \\ xaxis.z & yaxis.z & zaxis.z & 0 \\ -(xaxis \bullet eye) & -(yaxis \bullet eye) & -(zaxis \bullet eye) & 1 \end{bmatrix}$$

where:

- **zaxis = normal(LookAt – Eye)**
- **xaxis = normal(Up × zaxis)**
- **yaxis = normal(zaxis × xaxis)**

2.5.3.4. Screen Space

The **screen space** is a homogenous representation of the view space. The projection matrix is used to transform all objects from view space to screen space and map them to the

viewport[1]. There are two kinds of projection matrices. The first one is a default perspective matrix, while the second one is an orthogonal projection matrix, which flattens the Z axis and as a result allows representation of 3-dimensional objects in two dimensions.

Perspective projection matrix is built using the following parameters:

- **FovY -** the field of view in radians
- **Aspect** - viewport aspect ratio, defined as viewport's width divided by its height
- **Zn** - near view-plane, which cuts-off everything that is closer to the camera than the near view-plane
- **Zf** - far view-plane, which cuts-off everything that is further from the camera than the far view-plane

Knowing those values it is possible to build the perspective projection matrix:

$$\begin{bmatrix} \boldsymbol{xScale} & 0 & 0 & 0 \\ 0 & \boldsymbol{yScale} & 0 & 0 \\ 0 & 0 & \boldsymbol{Zf\ /(\ Zf - Zn\ )} & 1 \\ 0 & 0 & \boldsymbol{-Zn * Zf\ /(\ Zf - Zn\ )} & 0 \end{bmatrix}$$

where:

- **yScale = cot(FovY / 2)**
- **xScale = yScale / Aspect**

Orthogonal projection matrix requires a slightly different set of values, which include:

- **W -** width of the view volume
- **H** - height of the view volume
- **Zn** and **Zf**, which are the near and far clipping planes respectively.

The matrix can then be built using the following template:

$$\begin{bmatrix} \boldsymbol{2/W} & 0 & 0 & 0 \\ 0 & \boldsymbol{2/H} & 0 & 0 \\ 0 & 0 & \boldsymbol{1/(\ Zf - Zn\ )} & 0 \\ 0 & 0 & \boldsymbol{Zn/(\ Zn - Zf\ )} & 1 \end{bmatrix}$$

[1] A **viewport** is a 2D visual area, usually rectangular, onto which a 3D scene is projected.

2.5.4. GPU Fixed Rendering Pipeline

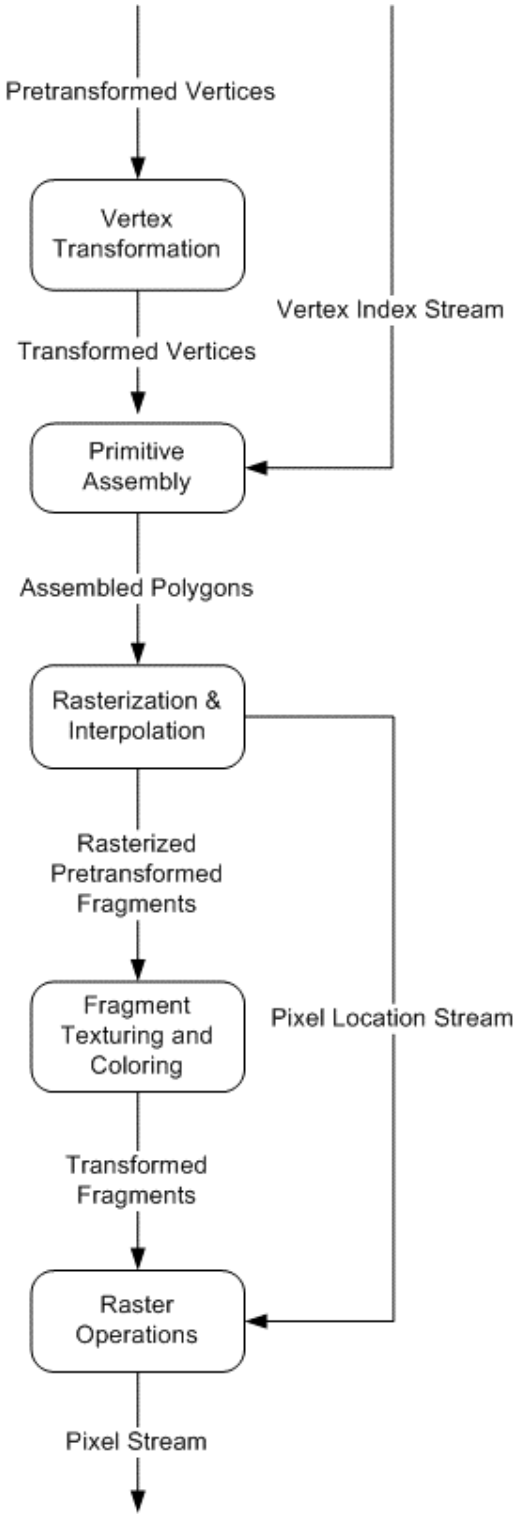

Figure 12: GPU's Fixed Rendering Pipeline

Hardware rendering is a sequential process *(see Figure 12)*, meaning that every step follows another always in the same exact order. A fixed pipeline refers to a standard rendering pipeline present in all graphics accelerators. It allows quick and simple rendering of scenes but is limited in what it can do as it only allows the programmers to set parameters that are present in the pipeline's specification. Any new effects or rendering techniques are impossible to achieve or require huge amounts of work to trick the pipeline into doing something it technically should not be supposed to do.

As its input, the rendering pipeline receives a set of vertices and information about their connectivity. The vertices can be connected into lines, polygons or be just a simple point list. The first step, however, is not shape assembly but **vertex transformation**. In this step the GPU uses the World, View and Projection matrices to transform the vertices from object space to world space, then to view space and to finally project the vertices onto the screen's viewport space. Additionally, texture coordinates are generated and colors applied to vertices after performing simple lighting on them. Such transformed vertices are then sent to **the primitive assembler**, which uses the connectivity information to form points, lines or polygons using the transformed vertices *(see Figure 13)*.

After all shapes are assembled the data is sent to the **rasterizer**, which performs several different

operations. First and foremost it removes all the triangles that are outside of the view frustum[1]. This operation is called clipping and not only does it remove all the triangles that are outside of the visible area but also those which are outside the area defined by the near and far clipping planes. Additionally faces that are facing forward or backward can be removed during this stage in a process called face culling. All triangles that remain after clipping and culling operations are rasterized into a form of pixel[2]-sized fragments. What it basically means is that all drawn primitives are broken into small fragments for each pixel that the primitives cover *(see Figure 13)*.

For every fragment a pixel location and a depth value is calculated and also a set of interpolated values such as colors, texture coordinates, normals, etc are generated. The interpolated values are calculated from the vertices that build the primitive, which was used to generate the fragment. It is important to emphasize is that a fragment is not a pixel and therefore is not guaranteed to become part of the final image. Depending on its depth value it may or may not be discarded in the final stage of the pipeline.

After all the primitives are rasterized and all vertex values are interpolated and assigned to the correct fragments, the **texturing & coloring** stage applies the final color values to the fragments *(see Figure 13)*. It is done by merging the interpolated color values and sampled texels[3]. Additionally at this stage the depth value of the fragment might be changed or the fragment itself can be completely discarded.

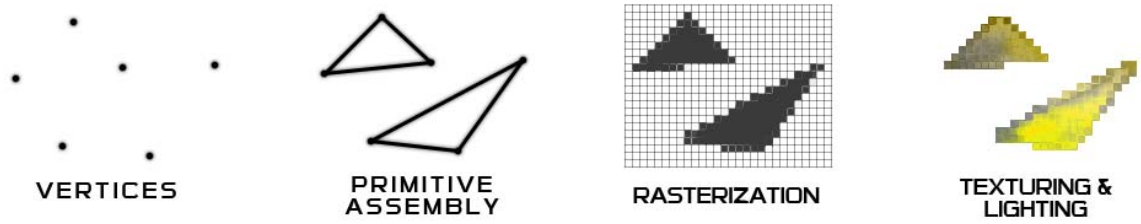

Figure 13: The visualization of the operations performed on the GPU

[1] A **view frustum** defines a region of space in the 3D world, which can be visible on the screen. It is a field of view of the camera in the 3D generated world. For perspective views it has a shape of a pyramid with a cut off top, while for orthogonal views it has a shape of a cuboid.

[2] A **pixel** is a singular unit in a graphic image. Each pixel contains 3 or 4 color values, such as red, green and blue components and additionally an alpha value.

[3] A **texel** is a singular unit (pixel) of the texture image.

The final stage of the pipeline performs **raster operations** before the resulting image is sent to the **frame buffer**[1]. During this stage several tests such as alpha, depth and stencil tests are performed. If any of those tests fail then the fragment is discarded.

The **alpha test** is performed when any of the fragments is transparent or semi-transparent and alpha blending is enabled by the programmer. If a fragment is allowed to be blended and passes the depth test then the color value of the final image pixel is blended with the new fragment color. The **depth test** is used to check whether any given fragment is not hidden behind any of the previous fragments that passed the test and occupy the same pixel location. A **z-buffer**[2] is used during the process as it contains all the depth values of all pixels updated during rendering. The **stencil test** uses a **stencil buffer**[3] to check whether the pixel that the fragment is assigned to can be updated. The stencil buffer is used in a similar way as an alpha mask. Depending on the color value of the pixel in the buffer a fragment will either pass the test or be discarded.

After all the tests are performed the final color values are assigned to correct pixels (and sometimes additionally blended) and finally sent to the frame buffer [Fernando03].

2.6. Programmable Shaders

Before the introduction of DirectX 8 all scenes had to be rendered using only the Graphical Processing Unit's Fixed Pipeline. This seriously limited programmers, designers and artists in what they could do. There was, of course, a certain amount of freedom in using the Fixed Pipeline and certain parameters could have been set and modified but no new effects or rendering techniques could have been achieved that were not supported by the GPUs at the time. This changed, however, with the introduction of DirectX 8 and Programmable Vertex and Pixel Shader units in the GPUs architecture. The so-called Shader Model 1.0 introduced a whole new range of possibilities. No longer were programmers forced to use the GPU's Fixed Pipeline. This time they were in charge of how a scene would be rendered. Usage of Vertex and Pixel Shader units required programmers to prepare small programs called Shaders, which would then be used by the GPU to determine the surface properties of the objects in a scene. Vertex Shaders would now need to contain information on

[1] A **frame buffer** is the final digital container of the image shown on the computer screen.

[2] A **z-buffer** contains information about the depth of each rendered pixel on the screen. Such information is used to determine which of the fragments mapped to the same pixel are closest to the camera's near plane.

[3] A **stencil buffer** is an 8-bit buffer, which is used to limit the rendering area by indicating which pixels can be updated with new fragments.

how to transform the vertices from Object Space to Screen Space, while Pixel Shaders would need to apply textures on the objects. Of course, this was not everything both shader units could do. Other techniques included per-vertex and per-pixel lighting, shadowing, normal mapping and many, many more.

2.6.1. GPU's New Programmable Pipeline

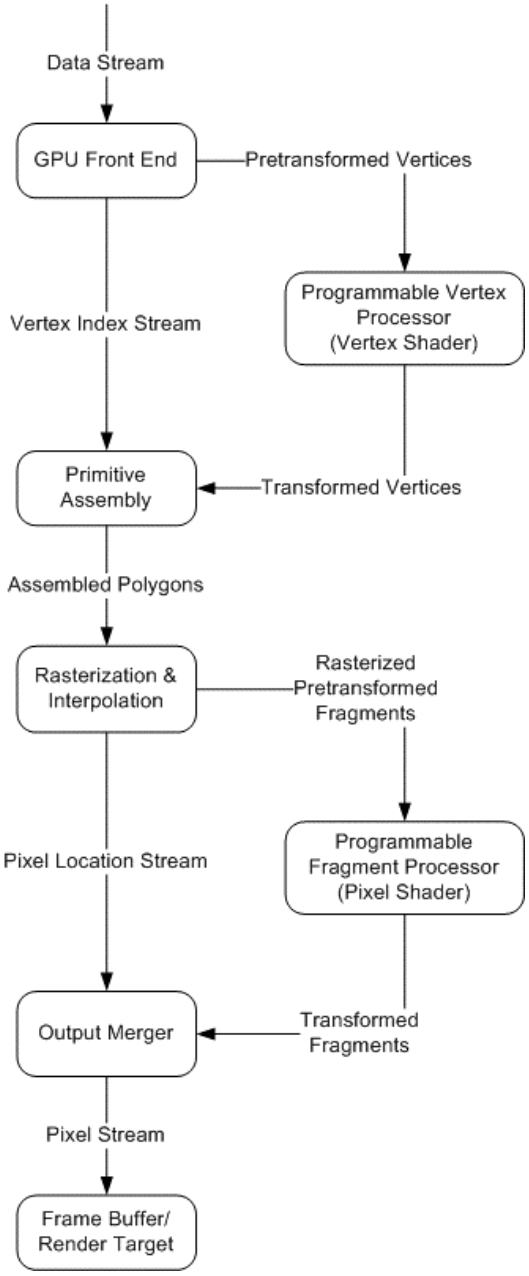

Figure 14: The schematic of the programmable graphical pipeline

As mentioned before, Vertex and Pixel Shader units became a part of the GPU's new programmable pipeline with the introduction of DirectX 8. Vertices became the responsibility of Vertex Shaders, while Pixel Shaders took care of pixel-sized fragments of the rasterized triangles *(see Figure 14)*.

Shaders have access to several constant registers which can be used to send additional information to the GPU (for example the WorldViewProjection matrix[1]), as well as a set of temporary registers which can be used at run-time to store results of complex operations that will be required later on. Because Vertex Shaders are invoked for each and every vertex and Pixel Shaders for each and every fragment that is part of the rendered object, it is a common notion to store pre-calculated information on the GPU and reuse it on each vertex and fragment instead of recaculating everything during each shader iteration.

As input, Vertex Shaders receive a list of pre-transformed vertices, which can contain information such as positions, texture coordinates, normal vectors, as well as skinning information required for animations. In other words all the information the vertex contains is received by the

[1] WorldViewProjection matrix is a matrix created by multiplying the World, View and Projection matrices.

Vertex Shader. From that point it is up to the programmer to decide what to do with that information. The output data, however, must consist of vertex positions transformed from Object Space to Screen Space and can also include texture coordinates for the Pixel Shader to use. Color values are also necessary if per-vertex lighting is used and the remaining output registers can be used to send data like normals or eye and light vectors.

Similarly to the Fixed Pipeline, once Vertex Shader completes its operations the transformed vertex positions are then used to determine the visible areas of the triangles, which are then rasterized into a set of fragments. The fragments are then sent to the Pixel Shader along with the input data – the interpolated result of the output data from the Vertex Shader, calculated based on the position of the fragment between the three vertices that form a triangle the fragment's part of. What Pixel Shaders have to do next is calculate the color values of the fragments by performing texturing and lighting calculations (if per-pixel lighting is desired) as well as shadowing. Then the new color values are sent to the output registers along with an optional new depth of the fragment. All the fragments are then sent to the Frame Buffer where the resulting render is finally assembled into a form of an image.

2.6.2. Limitations and Improvements

When it was introduced in the year 2000, the original Shader Model had many limitations. The main problem came from the instruction limit of both Vertex and Pixel Shader programs. While Vertex Shaders allowed for up to 128 instructions, Pixel Shaders were limited to only 4 texture calls and 8 arithmetic instructions. This forced programmers to divide the more complex calculations into several shader programs and render the scene's geometry in several passes. Different lights had to be rendered separately, shadows had to be applied in yet another shader and all that had to be also done strictly in GPUs assembly, as there was no High-Level Shader Language[1] available back then. With time, however, the instruction limit slowly rose with the introductions of updated Shader Models (1.1 to 1.4) but the true update came in December 2002 when DirectX 9 premiered.

With DirectX 9 came Shader Model 2.0 and HLSL. Thanks to the introduction of HLSL, Shader programming became a lot simpler and due to a higher number of allowed instructions in both Vertex and Pixel Shaders (256 in VS and 32 texture and 64 arithmetic in PS) many texturing and lightening techniques became a lot easier to achieve and many new became

[1] HLSL or High-Level Shader Language was introduced with DirectX 9 and is a language designed specifically to simplify and generalize the process of writing shaders.

possible (parallax mapping or soft shadowing for example). In the long run, however, the new shaders still suffered from similar limitations the previous model did. Longer Pixel Shader programs made more effects possible but at the same time still did not solve the multi-pass rendering problem and lack of true dynamic branching[1] forced developers to prepare different shaders for different set of lighting techniques and graphical settings.

Fortunately, since the release of DirectX 9 shaders went through one more update with the introduction of Shader Model 3.0 in August 2004. The update allowed for at least 512 instructions in both Vertex and Pixel Shaders and also upped the number of shader registers. Texturing, lighting and shadowing could finally be performed in one single pass without affecting the performance as much as 1.0 or 2.0 shaders did. Another great feature that Shader Model 3.0 introduced was dynamic branching. For example, one could detect the number of lights visible in a scene and then perform lighting operations only on the visible lights and while each frame might have a different number of visible lights the shader would still remain the same and would dynamically adjust to the scene requirements. Worth mentioning is that Vertex Shaders were finally allowed to sample textures, which made yet another technique called Displacement Mapping[2] (previously calculated on the CPU) possible to do on shaders.

2.6.3. The Future – Shader Model 4.0 and beyond

It would seem that Shader Model 3.0 was powerful enough but soon it will also become a thing of the past. With Windows Vista and DirectX 10 a new Shader Model 4.0 will be introduced. It will enable programmers to use a new programmable unit: the Geometry Shader. It will be available on the next generation of the graphics accelerators[3]. The new unit is placed between the Vertex and Pixel Shader in the new Programmable Pipeline. While Vertex Shaders performed operations on vertices and Pixel Shaders on fragments the Geometry Shaders will perform operations on primitives, which means that it will have access not only to the triangles (or lines or points) but also to the set of vertices that compose them.

Shader Model 4.0 will also introduce many new technologies and changes. First of all, the

---

[1] **Dynamic branching** is a way to decide the flow of the program at run time. Only the newest GPUs are capable of performing such an operation, however often with a visible performance penalty present.

[2] **Displacement mapping** is a technique, which uses a grayscale texture to move the vertices of an object along the direction of the normal vector.

[3] As of December 2006, NVidia's **GeForce 8800 GTS** and **GeForce 8800 GTX** were the first two cards released to support DirectX 10 and Shader Model 4.0. ATI's **Radeon R600**, which is to be released in 2007, will also provide support for the new technology.

GPU's Fixed Pipeline will be removed and as a result anyone using DirectX 10 will be forced to transform and light the objects using shaders. Another powerful feature is connected to the Geometry Shader itself – the Stream Output. So far the programmable graphics pipeline has moved in one direction – from pre-transformed vertices, via the Vertex and Pixel Shaders to the result in the frame buffer. But thanks to the Stream Output programmers will be able to send information generated by the Geometry Shader back to the Input Assembler where that information can be saved and/or processed again via Vertex and Geometry Shaders while at the same time the previous results can be sent to the rasterizer and rendered to the frame buffer. But the biggest feature of the new programmable unit is that it will allow generation of new primitives. Fur generation, stencil shadow extrusion, point-sprite generation from points and even collisions – all that and much, much more will now be possible to calculate on the GPU.

2.7. Dynamic Lighting

In the beginnings of the accelerated real-time graphics era, programmers and artists were limited only to the GPU's Fixed Pipeline lighting system. It was based on what is nowadays referred to as the Phong[1] lighting model but limited only to per-vertex lighting. While cheap and fast, it only worked well on a limited number of materials, which was the main reason why 3D real-time generated graphics looked fake and plastic. However, things changed with the introduction of programmable shaders as the Fixed Pipeline was no longer required to perform object transformations & lighting[2]. Finally it became possible to perform quick per-pixel lighting and light the scenes with as many lights as required within, of course, the limits of GPUs computational powers.

2.7.1. Per-Vertex and Per-Pixel lighting

There are two ways of calculating lighting on the objects' surfaces. One method referred to as Gouraud[3] shading [Gouraud71] or Per-Vertex Lighting *(see Figure 15)* requires the use of Vertex Shaders to calculate color values of the vertices. The resulting values are

[1] **Bui Toung Phong, PhD** (1942-1975), born in Vietnam, was the inventor of the Phong reflection model and Phong shading interpolation method, which are both widely used in computer graphics. [Phong73] [Phong75]

[2] **T&L** or **Transformation and Lightin**g is usually referred to in the context of the GPU hardware fixed pipeline. It refers to the operations of vertex transformation and fragment coloring on the GPU.

[3] **Henri Gouraud** (1944) is a French computer scientist, who invented the Gouraud shading method.

automatically interpolated along the edges of the polygons and then further interpolated along their surfaces. The resulting color values are then applied onto the visible fragments. However, while simple and fast, this method is very imprecise as it uses a very limited number of information to perform lighting on the objects. To achieve good lighting effects big flat surfaces need to be composed of a bigger number of vertices and faces.

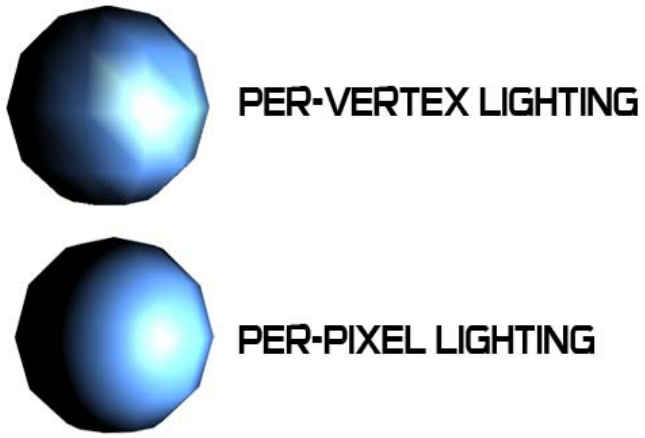

Figure 15: Comparison of Per-Vertex and Per-Pixel lighting methods

A more effective way is to use Pixel Shaders to calculate lighting and fragment colors. The method is called Per-Pixel Lighting *(see Figure 15)* and requires the whole lighting model to be calculated (and rarely for just one light) for each and every fragment [Lacroix05]. While more precise, this method takes a lot longer to calculate on the GPU. For a screen resolution of 1280x1024 that means performing complex calculations on at least 1310720 fragments each frame. Fortunately, nowadays GPUs can perform that many calculations (and more) per frame reasonably fast.

2.7.2. The Basic Lighting Model

There are many lighting models available, one for each different type of material. Each model has its unique properties and each one needs to be calculated in a different way. But one of the simplest and most popular ones is the Phong lighting model. As mentioned before, the Fixed Pipeline's lighting model was based on it and so it is convenient to use it as a basis for lighting explanations presented in this chapter.

It is necessary to explain how the basic lighting model works before various types of light sources are even mentioned. The most simple model requires that for each fragment four different color values are calculated, which are then summed into the resulting fragment color value:

$$finalColor = emissive + ambient + diffuse + specular$$

Figure 16: Render of the emissive component

The **emissive** component *(see Figure 16)* describes the light being emitted by the object's surface. It is especially useful when simulating glowing objects but because the value is the same for every fragment of the rendered surface the resulting image is flat and evenly colored with the emissive color value. Emissive surfaces should affect all nearby objects similarly to their behaviour in real life. However, in real-time generated images an emissive surface does not automatically equal a light source and as a result it does not automatically illuminate nearby surfaces and does not cast shadows. Hence for each emissive surface a point light should be created or simulated and the shadows should also be calculated and applied onto the affected surfaces. A slightly different approach to the emissive component requires the use of a glow texture, which allows to set different emissive values on different fragments.

Figure 17: Render of the ambient component

Ambient lighting *(see Figure 17)* refers to the light that does not have any particular source but rather comes from all directions. Ambient component describes such light reflected by the surface and is evenly applied all over the surface, similarly to the emissive light.

The ambient component is mathematically described as the surface's ability to reflect color (often used here is the surface's color value taken from its texture) multiplied by the ambient light's color value:

$$ambient = K_a * ambientColor$$

where:

- $\mathbf{K_a}$ refers to the surfaces reflection ability (or color)
- **ambientColor** refers to the environment's ambient lighting.

Diffuse *(see Figure 18)* is the third component of the basic lighting model described in this chapter. It describes the directional light reflected off the surface of the object in all directions. Calculating diffuse lighting allows the surface's irregularities and/or curves to become more visible. The amount of light being reflected depends on the angle in which the light ray strikes the surface.

Figure 18: Render of the diffuse component

Mathematically, diffuse lighting is described as:

$$diffuse = K_d * lightColor * saturate(dot(N, L))$$

where:

- **saturate** refers to a HLSL function, which for all values lower than 0 returns 0 and for all values higher than 1 returns 1.
- **dot** refers to a HLSL function, which returns a dot product of two given vectors
- $\mathbf{K_d}$ refers to the surface's diffuse color
- **lightColor** refers to the color of the light rays affecting the object
- **N** is the normal vector of the affected point of the surface
- **L** is the vector pointing from the affected surface point towards the light source's location

To acquire the diffuse value of the light reflection, it is necessary to calculate the dot product of the surface point's normal vector (N) and the light vector (L) *(see Figure 19)*. This allows the light source to affect the surfaces differently depending on the angle at which the light rays hits the surface. Because for angles higher than 90 degrees between the two vectors the dot product returns negative values, the result needs to be further saturated to ensure that the returned value stays between 0 and 1.

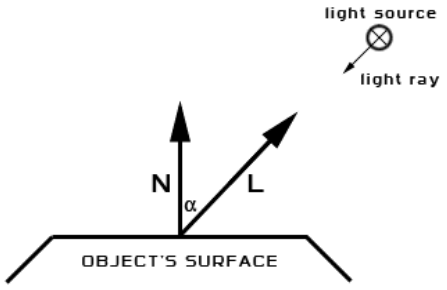

Figure 19: Visualization of vectors used in diffuse lighting calculations

The **specular** component *(see Figure 20)* takes into account the position of the viewer. The reason is that it describes the light reflected towards the viewer, which always appears brighter than the normal diffuse light. Specular highlight is mostly visible on shiny surfaces like metals and is more contained and tighter, whereas it is more spread out on less shiny materials.

Figure 20: Render of the specular component

Specular component is mathematically described as:

$$specular = K_s * lightColor * saturate(dot(N, H))^{shine}$$

where:

- $\mathbf{K_s}$ refers to the surface's specular color
- **lightColor** refers to the color of the light rays affecting the object
- **N** is the normal vector of the affected point of the surface
- **H** is halfway vector exactly between the light vector (L) and view vector (V)
- The higher **shine** factor the tighter and more contained the specular highlight becomes.

The halfway vector used for specular lighting computation needs to be additionally calculated from the view vector (V) and light vector (L) *(see Figure 21)*. The mathematical formula for calculating the halfway vector is:

$$H = V+L / |V+L|$$

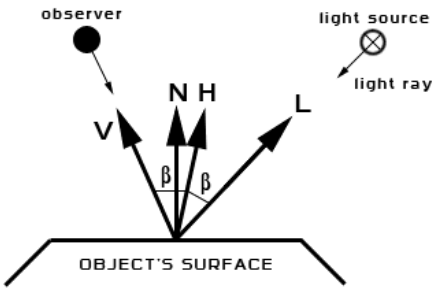

Figure 21: Visualization of vectors used specular lighting calculations

where:

- **H** is the resulting halfway vector in between vectors **V** and **L**

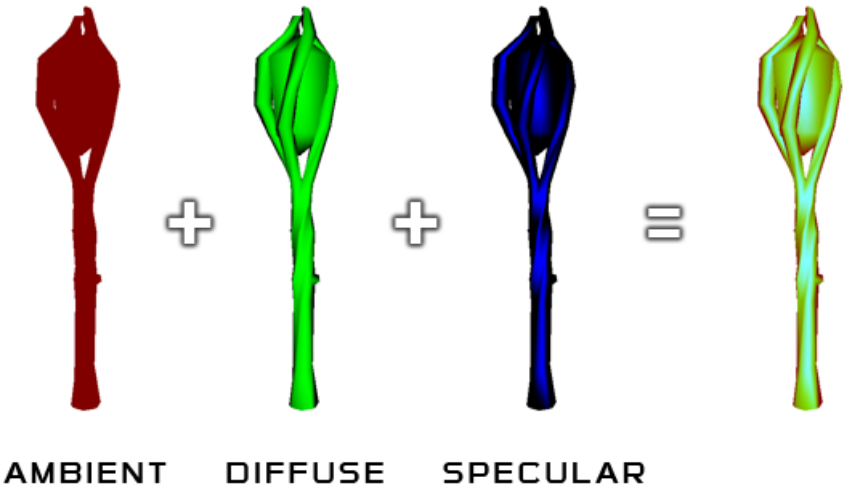

Figure 22: Blending of ambient, diffuse and specular components into the final image

2.7.3. Point-, Spot- and Directional Lights

The described lighting model *(see Figure 22)* is a very basic one and does not take into account many other factors that are used while lighting a scene. Many of the additional factors, however, are used to represent different types of lighting, which this section will describe.

The three light types supported by the Fixed Pipeline are point-, spot- and directional lights. Shaders require those light sources to be calculated manually, therefore this section will mainly concentrate on the mathematical representation of all three types of light sources.

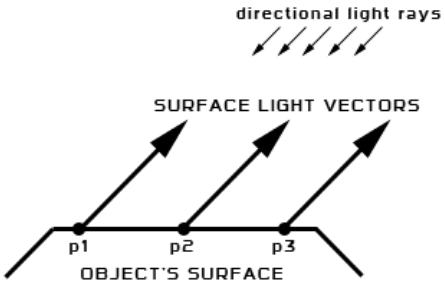

Figure 23: Directional light

Directional light *(see Figure 23)* is the simplest of all three concepts. It works exactly like the basic lighting model described on the previous pages with the difference being that the light vector stays always the same and does not need to be recalculated for every different fragment. A good example of a directional light source would be the Sun, which is located so far away from Earth that all light rays seem to come from the same direction. This is only a simplification since directional lights do not exist in nature but it is a very useful method of simulating Sun- or Moonlight in real-time rendered or pre-rendered scenes.

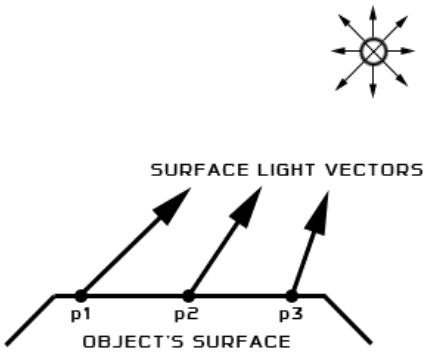

Figure 24: Point light

Point light *(see Figure 24)* is the second type of Fixed-Pipeline's light sources that can be calculated on a per-pixel basis (it can also be calculated on Vertex Shaders if required). Point Lights are omni-directional, which means they cast light rays in all directions. A good real-life example of a point light source would be a lightbulb.

The basic lighting model assumes that the light always has the same intensity no matter how far from the light source the object is. This approach is of course useful while simulating sunlight, but in real life the further an object is from a light source the less affected it is by it. Therefore, it is necessary to introduce a new variable called distance attenuation to affect the diffuse and specular lighting of the point light source. The attenuation factor is mathematically represented as:

$$attenuation = 1 / k_C + k_L d + k_Q d^2$$

where:

- **d** is the distance from the light source
- $\mathbf{k_C}$ is the constant coefficient of attenuation
- $\mathbf{k_L}$ is the linear coefficient of attenuation
- $\mathbf{k_Q}$ is the quadratic coefficient of attenuation

Having three values to fine-tune the amount of lighting attenuation allows for more precision than the real-life model, which is $\mathbf{1/d^2}$. The following mathematical formula demonstrates how attenuation fits in the basic lighting model's equation:

$$finalColor = emissive + ambient + attenuation * (diffuse + specular)$$

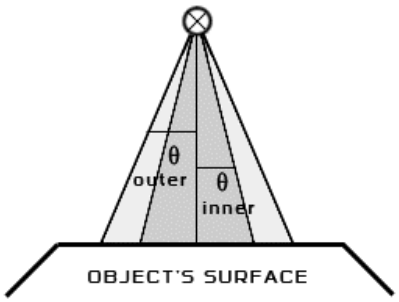

Figure 25: Spotlight

Spotlight *(see Figure 25)* is another commonly used type of lighting. It requires a light cut-off angle which controls how wide the spotlight cone is and how much of the object's surface is affected by it. Only fragments within the spotlight cone will be lighted.

To calculate spotlight lighting it is necessary to know the position of the spotlight and its direction as well as the position of the affected fragment. With this information it is possible to calculate the V vector (from the spotlight's position to the fragment's position) and D vector (the direction of the spotlight), both of which need to be normalized. A dot product of the two vectors is calculated and if it is bigger than the cosine of the cut-off angle then the fragment is affected by the spotlight.

This approach makes it possible to create spotlight effects. However, to make them look more natural, a second cut-off angle can be introduced. The first cut-off angle creates an inner cone, while the second one creates an outer spotlight cone. The idea is that the light's intensity would be uniform within the inner cone and would fade to zero intensity towards the outer cone's edge.

There are several improvements that can be added to spotlights. For example, instead of using a linear fade, a more sinusoidal one can be used. Additionally it is also common to attenuate the light's intensity depending on how far the light source is from the object, similarly to how it was achieved with point lights.

2.8. Texture Mapping

In computer generated graphics, additional detail on the object surfaces can be achieved using a technique called texture mapping. By applying bitmap images onto the surfaces of the polygonal meshes it is possible to simulate a greater amount of detail than the mesh itself actually presents.

2.8.1. Texture Mapping and Rendering

The 3D modelling software like Maya, 3DS Max or Blender all allow the images (called diffuse textures) to be mapped onto the object's surface *(see Figure 26)* in many different ways using for example, planar mapping, spherical mapping or torus mapping. As a result, the object's wireframe is laid out on a 2D UV map, which can be exported to a bitmap format consequently allowing artists to apply texture images onto it.

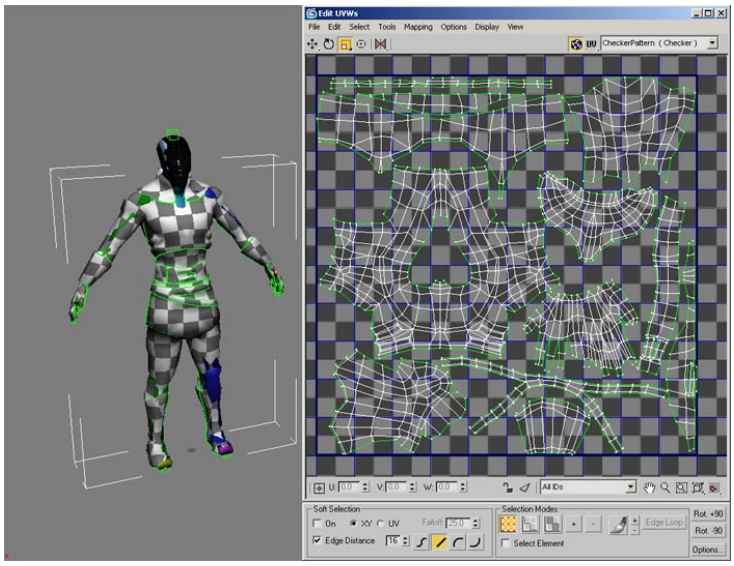

Figure 26: UV Texture Mapping in 3DS MAX

As mentioned before, each vertex of the textured mesh contains information about the texture's U and V coordinates. Those values allow the renderer to decide which texels should be applied onto the rasterized fragments. This of course can create some quality problems during rendering, as there may be situations when the fragment is much smaller than a texel, which can result in blurry renders. Similarly, fragments overlapping several texels require that the resulting colors value is interpolated between the texels applied onto a fragment, which can also affect the render's quality.

2.8.2. Texture Mipmapping and Filtering

One of the ways to improve the rendering speed and quality is to use a technique called mipmapping. A mipmap[1] is basically a chain of textures, each one with a power of two smaller resolution than the previous one. The highest mipmap level represents the original image and is used for objects that are close to the viewer while the next mipmap levels are used for objects that are farther away.

Another way to improve the quality of the renders and the way the textures are applied onto the objects' surfaces is to use different kinds of texture filtering. There are three types of filters that can be used on textures:

- **Minification Filter**, which performs filtering on fragments that are mapped to more than one texel.

[1] The name **mipmap** comes from a latin phrase *multum in parvo*, which means "much in a small space"

- **Magnification Filter**, which performs filtering on fragments that are smaller than a single texel they are mapped to.
- **Mipmap Filter**, which performs filtering on neighboring mipmap levels and improves blending quality of the transitions of the mipmap textures.

To each of these filters a different filtering algorithm can be applied. The most popular and widely used nowadays are Linear and Anisotropic filters and if no filtering is applied then a simple near-point texture sampling is used *(see Figure 27)*.

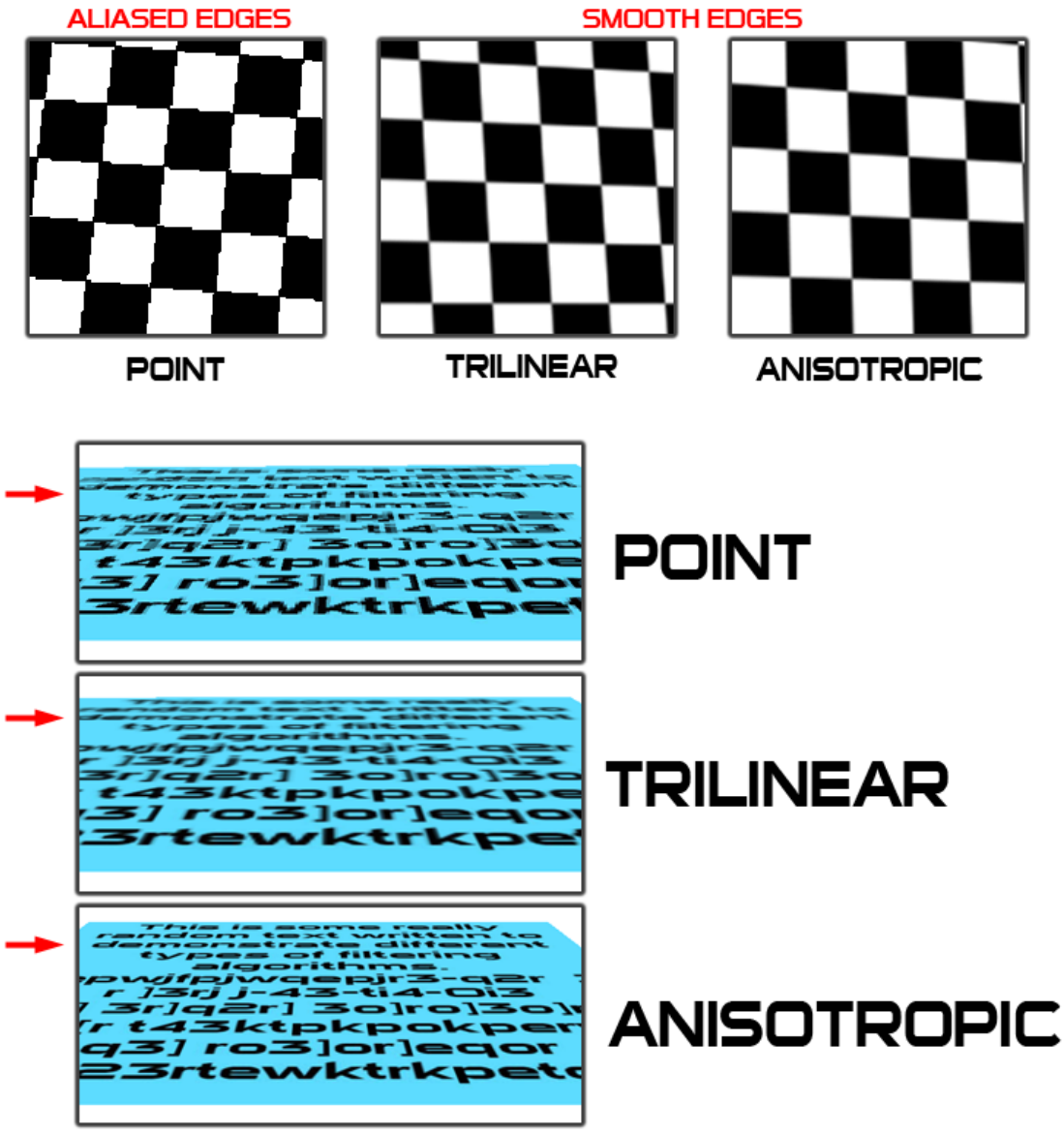

Figure 27: Comparisons of three texture filtering techniques: near-point sampling, trilinear filtering and anisotropic filtering

The **near-point texture sampling** method is fast and efficient but only if the texture's size is similar to the size of the rendered primitive on screen. Otherwise the resulting image may contain many artifacts such as blurring[1] and/or aliasing[2]. Near-point sampling translates the floating-point texture coordinates used in Direct3D, which range from *0.0* to *1.0*, to their integer equivalent ranging from *-0.5* to *n-0.5*, where n is the number of texels in the texture's given dimension. For texel coordinates that lie between the neighboring texels the closest texel is chosen. Those are the reasons for all artifacts that may be generated using this technique because even the slightest change in the calculated texture coordinates might cause a different texel to be applied than necessary.

Linear filtering solves all the problems presented by the near-point sampling method. What it essentially does is blend the neighboring texels depending on how close to the floating point texture coordinate value they are. There are two filtering techniques that use linear filtering: **bilinear** and **trilinear**. However, there is only a slight difference between the two. Bilinear filtering uses linear filtering on minification and magnification filters, while using a simple near-point sampling on mipmaps. This gives very good results when stretching or squashing the texture, but if mipmaps are used and the texture is applied on surfaces perpendicular to the plane of the screen then transitions between mipmaps will be more or less visible. To solve this, a linear filtering algorithm can be used as a mipmap filter resulting in a **trilinear filtering** technique. As a result, bilinear filtering is performed on two neighboring mipmap levels and the acquired values are then linearly interpolated into the final color of the fragment.

However, for the surfaces that are at angles with respect to the plane of the screen, both bilinear and trilinear filters result in blurriness on the distant parts of the surface. This is caused by the minification filter providing insufficient horizontal resolution of the textures. To solve the blurriness and preserve more texture detail, a different kind of filtering can be used: **anisotropic filtering**. With anisotropy the fragments mapped onto texels are distorted according to the view perspective. This results in the texture samples containing fewer texels from the distant parts of the textured surface and more texels from the closer parts. However, the extra quality comes at a high price. Since each anisotropic sample must be trilinearly or bilinearly filtered the number of texture samples rises greatly. For example, for anisotropy

[1] **Blur** refers to the appearance of an unfocused or unclear image.

[2] In computer graphics, **aliasing** refers to blocky appearance of diagonal lines in a bitmapped image caused by level of detail being higher than what the rasterizer can draw onto the viewport.

level of 16, 128 texture samples would need to be taken, as for each of the 16 anisotropic samples 8 extra texture samples would need to be taken to perform trilinear filtering. Fortunately, graphical accelerators nowadays have enough computation powers to handle anisotropic filtering with very good results.

2.8.3. Alpha, Specular and Glow Texturing Techniques

There are many different texturing techniques that can be used to increase the quality of the rendered scenes. Additional textures can be used in many different ways. They can either add more detail to the surfaces of the objects or they can help control the lighting in a more detailed way. Very popular (and simple) next to normal diffuse (color) textures are alpha, specular and glow textures.

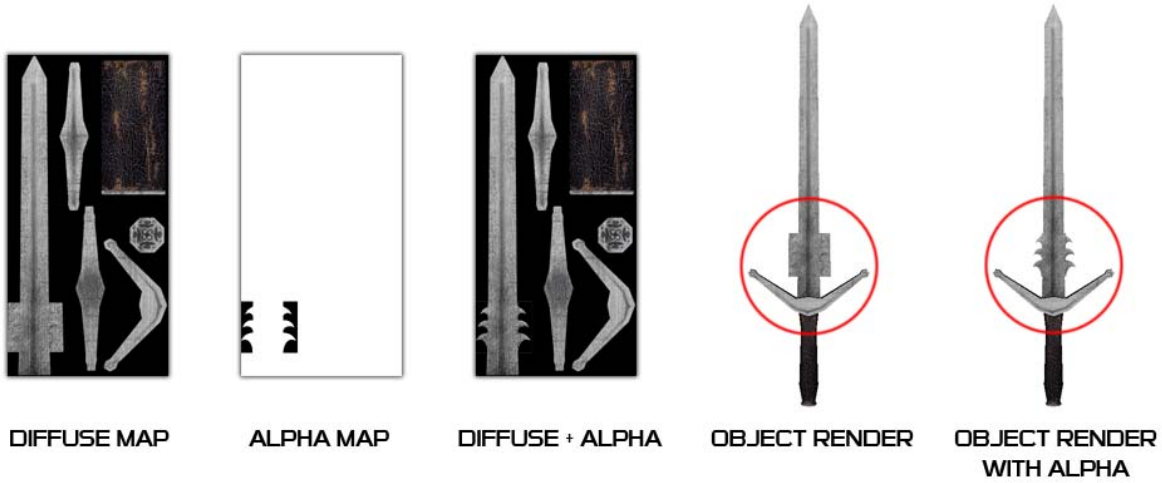

Figure 28: A sword rendered with applied diffuse and alpha textures

Alpha texture is basically a grayscale representation of the object's opacity. White areas of such a texture indicate areas that should be completely visible, while black indicate invisible areas. All grayscale values on the other hand make the object more or less transparent depending how close to white or black the color is. Alpha textures are very useful when more detail is needed to be generated from flat surfaces. Instead of modelling the details, they can simply be cut out of a primitive *(see Figure 28)*. Hair, curtains, windows and particle effects also benefit from the additional texture. To achieve transparent or semi-transparent surfaces, alpha-testing needs to be performed. If enabled on the GPU the test is performed automatically. This, however, requires the objects to be rendered in a specific order. Basically, alpha textured objects should always be rendered as last, because otherwise the blending will not be performed correctly and will result in visible render errors. The reason is that alpha blended objects take the color values of the already rendered fragments

that have a greater z-value and blend that value with the color of the alpha blended object. If objects that are behind the semi-transparent objects are rendered as second then chunks of those objects will not be visible because they will be cut-off by the z-buffer.

Figure 29: A club rendered with applied diffuse and specular textures.
Specular highlight is rendered with a blue color to more clearly indicate shiny areas of the object.

Specular texture can be either grayscale or color, depending on how the specular highlight needs to be controlled. If the highlight should use the diffuse & light color values only, then a grayscale texture is enough. But if it is required for the objects to shine in a completely different color, then a color specular texture must be used. Specular textures are extremely useful for objects that should only be partially shiny. For example, a piece of wood can have metal objects attached to it, like bolts or decorative surfaces. The wood should not be shiny and for that reason its specular highlight should be more spread out, whereas the metal elements should shine in the light with tight and contained highlights. A specular texture would allow to indicate which parts of the object are textured with shiny materials and which are covered with non-reflective surfaces *(see Figure 29)*.

If an emissive surface is present in a scene their emissive color can be controlled using the **glow texture** instead of using an uniform emissive value. While, of course requiring more memory to be stored, the glow texture increases the quality and realism of an object and can be used to achieve many interesting effects *(see Figure 30)*. For example, if a stained glass appears in a scene though which the Sun shines then the surface of the glass should be emissive to fake the Sun shining through it (raytracing[1] is not an option as it is still very

[1] **Raytracing** is a rendering method, which follows light rays from the eye point outward for each pixel of

expensive to compute in real-time). It is possible to place an emissive plane behind the window and mask it with a diffuse and alpha textured plane to fake the semi-transparent glass. Alternatively, a simple glow texture can be used on the stained glass object. Using the second approach it is possible to indicate the colors and places of the glass that should be emissive and those through which light would not shine through, such as borders between the glass elements. The visible effect is almost the same but with the second approach there is not only one less object to render, but also no object sorting and expensive alpha-testing is required.

Figure 30: A torch rendered with applied diffuse and glow textures

2.8.4. Normal Mapping

One of the more popular and widely used texturing techniques is normal mapping. It is a technique very similar to bump mapping as it also adds irregularities to flat surfaces simulating the extra surface bumpiness. However, neither bump mapping or normal mapping adds the details into the geometry itself and while low-poly edges are still visible, the surfaces will look just like a part of a high-poly model *(see Figure 31)*.

Why is normal mapping used in real-time generated scene and not bump mapping? The answer is that normal mapping is faster and easier to calculate. Both techniques modify the normals of the surfaces but the difference comes in the way it is being done. Bump mapping requires the normals to be recalculated for each fragment by checking the values in the grayscale height map, while normal mapping uses already pre-calculated normal values encoded in a form of a texture.

---

the render image's resolution. As a result it correctly simulates lighting, reflections, refractions and shadows.

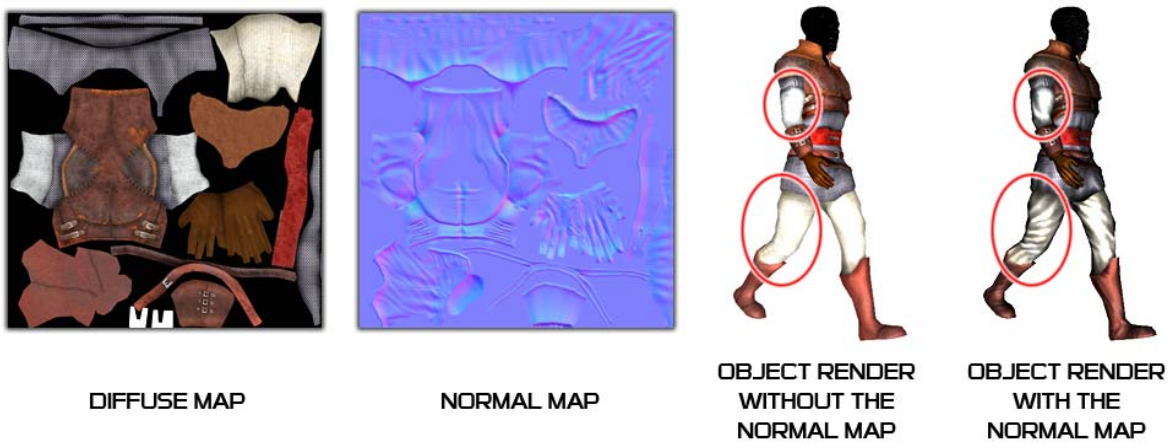

Figure 31: A guard avatar rendered with an applied diffuse texture and a tangent-space normal map

Normal maps are stored in a RGB[1] texture. Each of the color channels corresponds to a spatial dimension (X, Y, Z) which defines the direction of the normal. There are two types of normal mapping: tangent-space and object-space. The main difference between the two types comes in the way the normals are stored. Object-space maps store normals relatively to a constant world coordinate system, while tangent-space maps store normals based on a varying coordinate system derived from the local triangle space. In both types of normal maps, color channels represent the same data. The red channel is the relief of the surface when lit from the right, the green channel is the relief of the surface when lit from the bottom and the blue channel is the relief of the surface when lit from the front. For this reason object-space normal maps are very colorful, while tangent-space maps are mostly blue.

The more popular of the two are, however, tangent-space normal maps – even though they are more complex to calculate. There are several reasons for that. The main and most important one, however, is that tangent-space mapping allows for texture re-use on symmetrical or tiled parts of the model. The other reasons are more quality related and relate to better handling of mipmapping and post-processing as well as animations.

2.8.5. Parallax Mapping

Parallax Mapping uses a grayscale height-map to simulate more depth and irregularities on flat surfaces. It displaces the texture coordinates in such a way that depending on the view direction of the camera some texels become doubled and occlude other texels [McGuire05].

While normal mapping is capable of generating a certain degree of additional detail it still leaves the surfaces looking more or less flat. It is especially noticeable on surfaces like brick

[1] RGB – Red Green Blue – refers to the color representation of each pixel of the image.

walls or stone pavements where the texture is built from a series of bricks or rocks. Normally the area between the bricks would be a bit hollow making the bricks stick out slightly. A technique called Parallax Mapping makes such effects possible. As demonstrated in Figure 32 it adds an additional level of detail even to such a simple object like a brick road.

Figure 32: Parallax Mapping effect achieved using an additional height map

2.8.6. Other Texturing Techniques

There are many more texturing techniques than the ones explained so far. Each of those techniques has a different function and gives different results and many of them have been, or are starting to be used very extensively in games. Worth mentioning are:

- **Light Mapping**, which allows to add more detailed static lighting to a scene using a grayscale or color light texture.
- **Environmental Mapping**, which allows to simulate reflections of the surroundings on the shiny objects using pre-rendered cubemaps[1] of the environment.
- **Displacement Mapping**, which moves the vertices of a plane along a chosen axis according to a grayscale height map (especially useful when generating terrain meshes).
- **Shadow Mapping**, which allows creation of static or real-time generated dynamic shadows to the scene. The next chapter will explain this technique extensively.

[1] A **cubemap** consists of 6 textures placed next to each other forming a schematic of an unassembled cube.

2.9. Shadow Mapping

Shadows enhance computer generated scenes by adding more depth and detail to them. However, while relatively simple to achieve in raytraced renders, they are very complex to generate in real-time. At first, simple techniques like light mapping were used to simulate shadows and changes in the lighting on objects. However, with time and growth of the computation power of the GPUs, it became possible to apply different, more detailed and realistic techniques. Most importantly though, it finally became possible to generate real-time dynamic shadows, which could change depending on the light changes and movement of the objects.

2.9.1. Shadow Volumes

There are two shadowing techniques which became widely used and popular: shadow volumes and shadow maps. Both of the techniques differ greatly in how they are computed, their complexity and in the detail of the shadows they generate.

Figure 33: A DirectX sample demonstrating shadow volumes [DirectX1]

Shadows volumes *(see Figure 33)* require additional geometry being generated to perform shadowing. To do that it is necessary to extend all silhouette edges (such that are on the border of the surfaces that are in light and surfaces that face away from the light) of the affected objects in the direction away from the light's position. The generated rays are then capped by adding a front-cap and back-cap, which are basically faces which connect all the rays together at their beginning and desired end. This generates the so-called shadow volumes, which help divide the scene into two areas: the area that is in light and the area that

is shadowed. It is important to mention how this process is being performed if the differences between the two shadowing techniques as well as advantages and disadvantages of both are to be clearly explained.

To generate shadows using shadow volumes a scene must first be render as if completely in shadow. Then, after disabling z-buffer and frame-buffer writes, the back-faces and later front-faces of the shadow volumes are rendered and the depth values of both are compared with the values in the z-buffer using John Carmack's[1] method called Carmack's Reverse (or Depth Fail). If a rendered back-face of the shadow volume fails the depth test the stencil value is incremented and for each front-face that fails the depth test the stencil value is decremented. This way the stencil buffer becomes a sort-of a mask, which is used to apply lighting to the scenes. The whole scene (without shadow volumes) must be rendered once again, this time with all lighting effects on. As a result areas masked by the stencil buffer will remain dark, as if in shadow.

The pitfalls of this technique become apparent when shadows need to be generated for moving objects (such as avatars) or moving light sources. Currently shadow volumes need to be calculated strictly on the CPU (Shader Model 4.0 will allow generation of shadow volumes on the GPU using Geometry Shaders), which for very complex objects or a bigger amount of them might seriously affect the render engine's performance. Another problem comes from using non-convex meshes, which require additional computation to ensure that the shadows are rendered correctly. It is also worth mentioning that for this technique to work correctly all objects should be 2-manifold, which means that the entire mesh must be closed and each edge must be part of exactly two faces.

2.9.2. Shadow Maps

The alternative to shadow volumes are **shadow maps**, which help create shadows using an entirely different approach. Because shadow maps are completely texture-based, this technique does not require any additional geometry to be generator, nor any stencil buffer writes being performed. To generate shadows using this methods, the scene needs to be rendered from the light's perspective saving depth values of all visible surfaces into a texture. After that, the scene is rendered once again from the eye perspective, while comparing the

[1] John Carmack is an American game programmer, co-founder of **id Software**, a computer game development company. Carmack is credited for such games as Wolfenstein 3D, Doom and Quake and for many revolutionary programming techniques used to this day by many other programmers.

depth of every fragment drawn (as if it was seen by the light, rather than the eye) with the depth stored on the map. If the depth of the drawn fragment is bigger than the depth stored in the texture then the area is in shadow *(see Figure 34)*.

Figure 34: A location from the game rendered with dynamically generated shadows

Shadow maps are simpler, easier to compute and in most cases faster to generate and render. However, the speed and simplicity comes at a price. Whereas shadow volumes always generate crisp-edged and very detailed shadows, the quality of shadows generated with shadow maps depends mostly on the resolution of the depth texture. Shadow maps need to envelop the whole visible area and as a result, for large open spaces, they will generate very aliased shadows. Additionally, extra memory is needed on the GPU to store the shadow maps. But there is one thing that is only possible with shadow maps and that is alpha mapping. To put it simply, shadow mapping allows alpha mapped surfaces to cast alpha mapped shadows so if, for example, a shape is cut out from a single primitive using an alpha map that very shape will be the shape of the shadow. For obvious reasons it would be impossible to achieve using shadow volumes, since only the primitives would be used to generate the shadows.

2.9.3. Uniform Shadow Map Calculation

To apply shadows using the shadow mapping technique it is necessary to go through several computation stages. As mentioned before, first the whole scene must be rendered from the light's perspective, which means every object in the game must be transformed first from World Space to light's View Space and then to light's Screen Space (which will be called

Light Space from now on) using a light projection matrix. Setting up the light's view matrix is done in the same way as setting up a view matrix for the camera. In truth, very often lights are already implemented as cameras as the light's position and direction vector can used as parameters for camera's position and view direction. The next few steps, however, depend greatly on the type of the light source used. For both spot and point lights a standard perspective projection matrix is used but for directional lights an orthogonal projection matrix must be used. The reason for that is that, while spot and point lights generate light rays with the light's position as their point of origin, directional light rays are always parallel to each other and have the same direction.

Figure 35: Sian, the game's main character, rendered using dynamically generated shadow maps

Once both matrices are calculated and all the objects are transformed to the light's screen space, the z values of all the fragments are rendered to a floating-point texture, which becomes the light's depth map. It is important to mention that while for directional and spot lights one depth map is usually enough, point lights require exactly 6 depth maps being generated. The reason is that point lights generate light rays in all possible directions and because of that a cubemap is needed to ensure depth maps are generated for all possible objects the point light might affect.

The next step is applying shadows onto the scene. It can be done in two different ways. One way requires the shadows to be pre-generated first and rendered to a texture, which can later be blurred and applied on the fully lighted and rendered scene. The other allows shadows to be generated at the same time the scene is. It basically depends on the used Shader Model and the amount of effects that are applied during each frame. However, independently of the approach used, the shadows are generated the same way. To render shadows all objects must be transformed to camera's view space to generate fragments that will be visible on the screen, while at the same time transforming all the objects to Light Space and saving the resulting positions in one of the Vertex Shader's output registers. Thanks to this operation the Light Space position of each visible fragment will be known. In the Pixel Shader it is then necessary to compare the z

values (or depth values) of the fragments in Light Space to the depth values of the rendered objects as seen by the light source. Should the depth value of the fragment be greater than the one saved in the shadow map in the fragment's position then the fragment is in shadow and therefore should not be affected by the light source.

2.9.4. Shadow Map Problems and New Techniques

As mentioned before, the quality of shadows generated using the shadow mapping technique greatly depends on the resolution of the shadow maps. For very large view areas this can lead to serious aliasing problems and also to the so-called z-fights. Shadow maps which are rendered for very large areas are greatly inaccurate and can result in incorrect depth values for certain areas. This can result in some fragments to be shadowed when they are in light and vice versa. One of the very common effects generated by the inaccurate depth comparisons is the Moire-like effect *(see Figure 36)*, which can be to some extent corrected by adding a small biasing value to the depth values stored in the shadow map. In the long run, however, the inaccuracy problems and aliasing cannot be solved that easily.

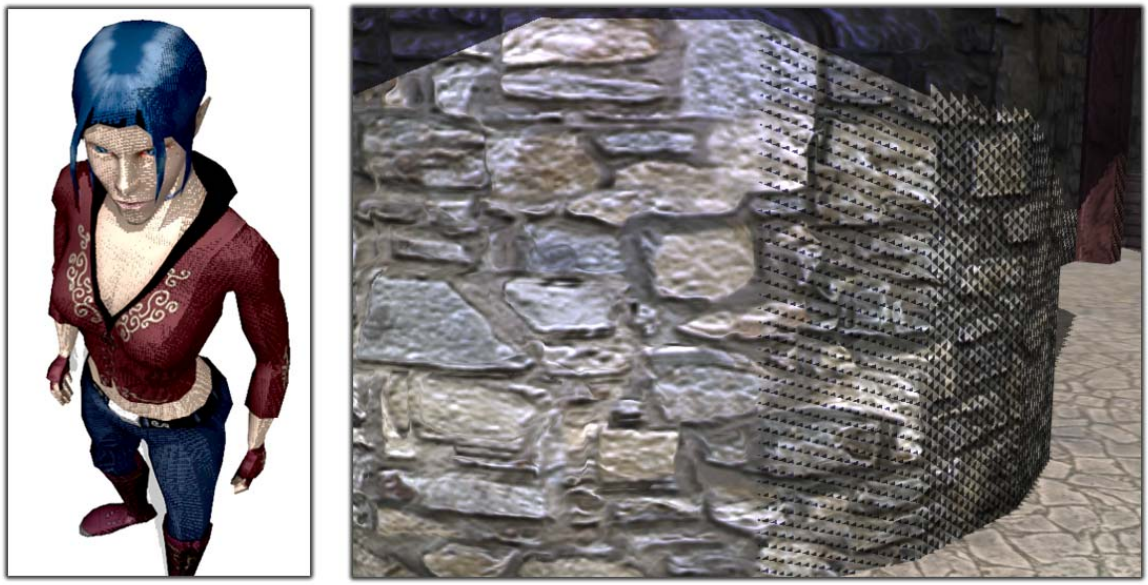

Figure 36: Moire-like effects resulting from incorrect biasing of the shadow map values

For many years since the first time shadow maps were used in games, programmers struggled to come up with ways to enhance the quality and accuracy of the shadows. The first alternatives to Uniform Shadow Mapping were the following techniques: Perspective Shadow Mapping (PSM) [Stamminger04], Trapezoidal Shadow Mapping (TSM) [Martin04] and Light Space Shadow Mapping (LiSPSM) [Wimmer06]. All three of those took advantage of a single idea. Uniform shadow maps generated the same amount of depth information for close and

faraway objects not taking into account their distance from the camera. As a result, the quality of the shadows was identical everywhere on the scene. Therefore, it was proposed that the perspective of the light's view could be warped giving more texture space for closer objects and less texture space for faraway objects. This generally improved the quality of shadow maps greatly. However, in some cases PSMs, TSMs and LiSPSMs generate even more aliased shadows than Uniform Shadow Maps. The problems are especially visible when the camera is facing towards the light source – no perspective warping is able to correctly allocate more texel space for objects closer to the camera in such a case. That is why several other techniques were developed such as Cascaded Shadow Maps (CSM) [Futuremark06], which generate a series of shadow maps for the whole camera view frustum. The shadow maps closer to the camera have a bigger resolution and cover a smaller area and as a consequence generate high detailed shadow maps. The next shadow maps are smaller and bigger, as less precision is necessary for further objects. Thanks to this method the whole scene can be shadowed with very high quality shadows at the cost of rendering speed and additional texture memory.

2.10. Post-Processing Effects

In many graphical programs it is possible to modify an already rasterized 2D image using a set of filters, transformations and effects. An image can be scaled, rotated, skewed or filtered in many different ways for example: blurred, darkened or sharpened. The range and variety of different possible effects is enormous and the results of such modifications can greatly improve the quality of the resulting image. It is no surprise then that such techniques are also used in real-time rendering applications. Such techniques as commonly referred to as image-space post-processing effects.

2.10.1. Light Bloom Effect

As mentioned before, image-space post-processing effects are performed on the already rasterized 2D renders of the scene. What it basically means is that before any effect can be applied the scene needs to be rendered directly to a texture rather than to the frame buffer. Once that is done, however, it opens a door to a wide range of effects. One of the more popular nowadays is a glare effect called **Bloom** or **Light Bloom**. It creates a sort-of bleeding effect around the brightest pixels of the image giving the user a more detailed information about the relative brightness of the different parts of the scene *(see Figure 37)*. Thanks to Bloom it is possible to achieve an effect when a background light source bleeds over onto the

objects in the foreground. It also becomes possible to simulate light sources brighter than what the monitors can display.

Figure 37: Render of the same scene with and without the Bloom effect

In the simplest of ways, the Bloom effect can be achieved by multiplying the colors of the pixels of the rendered scene by themselves, which makes the light areas brighter and dark areas darker. The results of such operation is then blurred and then blended with the original render of the scene. However, there is a small downside to the effect, which comes directly as a result of the transformations that are performed on the image. Applying a blurred image onto the original render causes very bright objects to become softened and consequentially losing some of their detail.

Bloom effects are very often used alongside High Dynamic Range Rendering (HDRR). The reason for that is that while Low Dynamic Range (LDR) allows for colors to have values ranging only from 0.0 to 1.0, HDR allows colors to have values above 1.0, therefore allowing for extremely bright light sources. Blurring an LDR image would cause the pixels with values higher than 1.0 to be treated as white and be blurred using a white-black gradient. HDR, on the other hand, retains the real color values of pixels and creates gradients with correct color values.

2.10.2. Gaussian Blur

Bloom effect requires the multiplied image to be blurred before it is applied onto the original rendering. Blurring allows the halo effect around bright objects to become visible and that is why it is very important to be able to perform it in a fast and efficient way in real-time.

A way to do it is to use the Gaussian distribution:

$$G(r) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-r^2/(2\sigma^2)}$$

The above equation makes it possible to calculate a NxN matrix with weight values which are then used to perform blurring. The color value of every pixel is multiplied by the weight value in the middle of the matrix and then neighboring pixels' color values are added to that result after being multiplied by the neighboring weight values. Because the value in the middle of the matrix is always the highest one Gaussian Blur allows for a much higher detail preservation level than simple blurring techniques, that average the neighboring pixel values.

Gaussian distribution has one very important property. Since the resulting value matrix is circularly symmetric it can be applied to the image in two distinct rendering passes: one horizontal and one vertical. Thanks to this, blurring can be applied much faster than if it was done both horizontally and vertically in one pass. Instead of calculating NxN weighted color values for each pixel only N+N values are calculated – N values in one pass and N values in another.

2.10.3. Motion Blur

Just like Bloom, **Motion Blur** adds an additional degree of realism to the rendered scenes. It is an inherent effect in photography and film as it appears when the objects or the camera move during the exposure of the image. The effect makes itself visible as a sort of blur along the direction of the object's movement. While often, though not always, undesired in photography it is ever-present in video recordings where it creates an illusion of smooth motion. However, such effect is not automatic in real-time generated sequences and that is why with a standard film framerate of 25-30 fps they might be perceived as 'jumpy' or not smooth enough. The reason is that all real-time images are generated using an exact moment in time and not during a short period of time as with photographs or videos. This causes renders to have no motion blur what-so-ever.

Motion Blur can be used to simulate high-speed movement, such as driving a very fast car or flying in a jet. In real life the environments visible from such vehicles during high-speed movement would be perceived as blurred as they would move past the driver and the vehicle at very fast speeds. So Motion Blur would allow to create more realistic visualizations of such examples. Of course, Motion Blur may be used to create other effects than just high-speed movement. If applied correctly it can help create slow motion sequences or

dizziness/drunkenness effects.

As with each technique there are several ways and approaches that can be used to create the Motion Blur effect in real-time. The simplest and most demanding way in terms of computation powers is rendering the scene several times in every frame and then blending all renders together. Each time the scene would be rendered it would be updated by a fraction of the update time. For example, if 5 renders are used to create the motion blur effect then each render would be updated by $1/5^{th}$ of the elapsed time between the two consecutive frames. All renders would not be blended evenly. The first render would receive the smallest blending weight, while the last render would receive the highest weight. Such method, while effective, is greatly inefficient. Not only does it require all of the geometry being rendered several times, which for very complex worlds can greatly decrease the performance, but also, in some cases, may require several render targets being used to store the consecutive renders.

A different approach is presented in one of the samples shipped with DirectX SDK [DirectX2]. It uses only two render targets to perform Motion Blurring. The whole scene is rendered to one RGB texture, while at the same time the objects' velocities are saved using new and previous frames' transformation matrices to calculate the difference in pixel positions. Then in a second pass an image-space post-processing effect is being performed by taking the velocity values and sampling and averaging several texel values along the direction of the object's velocity.

2.11. Particle Effects

Particles are used to generate physically realistic randomized effects, such as smoke, sparks, blood and rain. A particle is calculated as a point in space, but can be represented as a mesh or just a textured quad[1] that is facing the camera. Every particle goes through a complete (or almost complete) life cycle: it is born or generated from a certain point or plane, its parameters are changed over time and then, if not needed anymore, it is destroyed.

Particle effects are created by generating a set of particles from either a single point in space or an area defined by the surface of a plane or a mesh. Different effects are created by applying different mathematical transformations to the particle's parameters, such as color, position or size. Since particles are meant to behave realistically, they are programmed to interact with the game's environments. For example, a rain effect can be set up to react to an in-game wind or explosions.

[1] Quad is a rectangular plane consisting of four vertices

Nowadays, particle effects are used in almost every game created. Their applications include generation of smoke, fluids and weather effects as well as lightning effects and fire.

2.12. Sound

One of the main elements of every game is sound. It enriches the game's realism and immersion and most importantly the gamer's experience. Without sound, games would feel empty and so it is important to take great care when designing the engine's sound managing system and creating the sounds. A study done by LucasFilm[1] during the testing phase of the THX[2] standard proved the importance of high quality of sound. A group of people was shown two identical movies both of which differed only in sound quality. The result was surprising as it turned out that movie with higher sound quality seemed to be much sharper and clearer.

2.12.1. Speaker Systems

A good sound speaker system is requires to achieve good sound quality. The most common speaker configurations for home PCs are stereo speakers and 5.1 speaker systems[3]. While stereo configuration is more than enough for stereo music, its limits become apparent when true 3D sounds are being played. With just two speakers it is almost impossible to simulate the sounds that come from behind the player. Therefore a 5.1 speaker system was invented, which allows for almost seamless 3D sound playback. The only drawback comes from the fact that sounds coming from below or above the player cannot be played in true 3D as two more speakers would be required to achieve such an effect.

Another important piece of hardware is the sound card. Its purpose is to transform and interpolate the received raw sound samples and send them via output to the speakers (or headphones for example). Because a 3D sound has a 3D origin and is specialized for four speakers, there is a number of interesting effects that can be achieved by the sound card. Some of those effects include echo, reflections, Doppler shifts, pitch bending and much more.

---

[1] Lucas Film is a film production company founded by George Lucas. It is best known for producing the Star Wars movies. It also has been a leader in new technologies for film creation in fields of special effects, sound and computer animation.

[2] THX is the name of a sound reproduction system for movie theaters, computer speakers and gaming consoles.

[3] 5.1 is a format for surround sound which uses six speakers: center, front-left, front-right, rear-left, rear-right and subwoofer.

2.12.2. DirectSound and DirectMusic

There are many sound APIs available to make sound programming easier. The ones bundled with DirectX are DirectSound and DirectMusic.

DirectSound provides the interface between the application and the sound card, making playback of 2D and 3D sounds as well as music very quick and easy. It also provides many functions such as audio mixing and volume control and effects like echo, reverb, and flange.

DirectMusic provides a more high-level control over sound playback and some of its features overlap with those of DirectSound. Despite its name, DirectMusic plays all kinds of sounds but its main feature is MIDI[1] and dynamic music playback. Thanks to dividing music into sections it is possible to change the order of the sections dynamically and modify music playback depending on what is going on in the game. Sections can be speeded up, blended and all that allows for a more responsive music in games. Being a high-level API, DirectMusic generates a much higher latency during playback than DirectSound and for that reasons in applications requiring fast and seamless sound playback DirectSound is used.

During the time this document was being written (November 2006) DirectMusic was already a deprecated API for some time. It was, however, only the first step in changing the set of DirectX APIs. The introduction of The Microsoft Cross-Platform Audio Creation Tool (XACT) will cause DirectSound to be deprecated as well when DirectX 10 premieres.

XACT was originally developed for Microsoft's XBOX console but was soon moved over to DirectX and now also XNA[2], allowing for cross-platform (PC and XBOX) audio creation and programming. The main feature of XNA is that it allows binding a set of sound files into one file called a wave bank and another file called a sound bank, which contains information on how to playback bundled files in the wave bank. XNA also provides a very powerful feature called zero-latency streaming. Normally, to play a sound file it would be necessary for the storage device to place the read/write head at an appropriate sector before the file would be played. For hard drives it can take as long as 100 milliseconds (worst-case scenario), but CDs or DVDs may sometimes require even a couple of seconds to perform such an operation. XACT, however, solves that problem by strategically allocating small portions of audio files

[1] MIDI is a protocol that enables electronic musical instruments, computers (and many other equipment) to communicate, control and synchronize in real time. MIDI simply transmits digital data such as the pitch and intensity of musical notes to play and additionally controls parameters like volume, vibrato and panning.

[2] XNA is a game development framework developed by Microsoft the purpose of which is to simplify and encourage the creation of independent games.

in small-sized buffers and then loads the subsequent portions of the file as the buffers become free again. Thanks to such approach it is possible to playback sound files without any latency what-so-ever.

2.12.3. Sound formats

When designing a sound managing system for games it is important to consider the format of the sound files that are going to be used. There are currently three most widely used formats: **WAV**, **MP3** and **OGG Vorbis**.

WAV (Waveform Audio Format) is a standard Microsoft and IBM file format for storing audio. Although it can hold compressed audio the most common WAV files contain raw, uncompressed data in the form of pulse-code modulation (PCM). PCM is the standard audio file format for CDs at 44,100 samples per second. Because of that, WAV files provide maximum sound quality at the price of taking up more storage space.

MP3 (MPEG-1 Audio Layer 3) is one of the more popular audio encoding formats. While providing a lossy compression it still produces relatively high quality sound compared to the size of the files. MP3 is one of the formats widely used in games but slowly more and more companies decide to use **OGG** as their format of choice.

OGG is not an audio file format in itself but rather an open multimedia container format designed for efficient streaming and manipulation. It can contain a number of different data types alongside audio, such as video and text. Most commonly in games, however, OGG is used to store audio data encoded into the Vorbis format. The main advantage of the OGG Vorbis format is that it is completely open-sourced and patent free (which is not the case with the MP3 format). Thanks to that OGGs can be used freely without any charge in any way required. Studies have also shown that the OGG Vorbis format provides slightly higher quality audio than MP3s.

In conclusion, it is worth mentioning that while **MP3** and **OGG Vorbis** formats provide a good trade-of between file sizes and sound quality, both are eventually streamed to a compressed WAV format during playback in real-time applications.

2.13. Space partitioning

Modern games have a complex and sophisticated geometry, which forces the engine to perform a huge amount of calculations on all objects. This can cause the performance to be heavily affected if the geometry is not additionally managed. To optimize rendering speeds, as well as other elements of the engine, a system that partitions the 3D space must be

implemented. That system must provide functionality for detection of visible geometry so that only visible areas are rendered. It can also provide a way to divide the game's world into collision sectors allowing objects to collide only with the nearest environment.

2.13.1. Space Division Basics

There are many data structures and algorithms that help create such a system. However, there are only three most common data structures used in space partitioning: binary space partitioning (BSP), QuadTree and OctTree. Those three data structures differ only by number of planes that divide the 3D space. For BSP it is recursively one plane, for QuadTree it is recursively two planes and for OctTree it is recursively three planes. The process of creating the space division trees is very similar in all three cases. The only difference comes from the number of children each node can have. For OctTrees it is eight children, for QuadTrees it is four children and for BSPs it is two children [Ginsburg00] [Ulrich00].

2.13.2. Building the Space Division Tree

The process of creating a space division tree is very simple. The first step is to gather all the geometry that is to be partitioned by the tree. The second step is to generate the root of the tree, which will enclose all of the geometry. Then the root node is recursively divided (each new child node is further divided using the same algorithm) until user defined conditions are met (usually either the amount of vertices in a node or number of objects or a maximum number of divisions). When new nodes are created, new surrounding cubes are generated that contain only parts of the world's geometry. Eventually each node of the tree contains:

- A cube surrounding the geometry
- World's geometry or list of objects belonging to the node
- Children, which are smaller nodes created by dividing the root node.

2.14. Collision Detection

Collision detection is a very important part of all 3D games. A game must have a collision detection system that will provide enough functionality for objects to interact with the world and the characters correctly. It is a vast subject and there are many solutions to it. Explained below are a few basic methods on which more sophisticated algorithms are based [Kaiser00].

2.14.1. Bounding-sphere Collisions

The simplest of collision detection methods uses bounding-spheres. A bounding-sphere requires a calculation of two values: the object's center point and the sphere's radius. The first

value is found by averaging the positions of all of the object's vertices and the radius is achieved by calculating the distance between the object's center and the most distant vertex from the center.

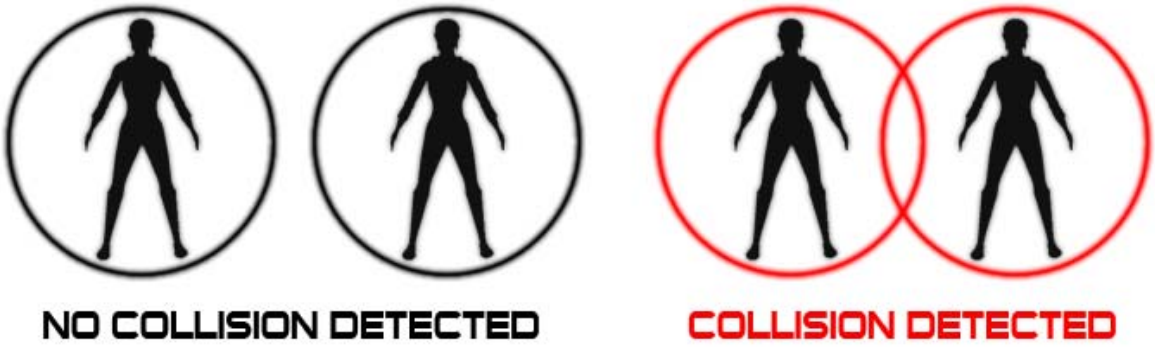

Figure 38: Bounding-sphere collision detection

To detect a bounding-sphere collision all that has to be done is to measure the distance between the two objects and compare the result with the sum of their bounding-sphere radiuses. However, since the bounding-sphere collision detection is very inaccurate it is most often used only to determine if there is a possibility of two objects colliding with each other.

2.14.2. Bounding-box Collisions

A more accurate method for collision detection requires the use of bounding-boxes. A bounding-box is described by two points: the minimal one and the maximal one, which are calculated by finding the smallest and biggest X, Y and Z position coordinates of the mesh's vertices.

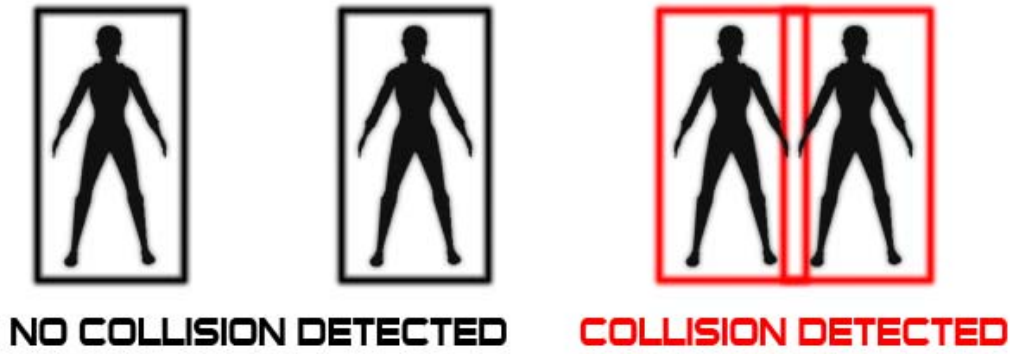

Figure 39: Bounding-box collision detection

There are two kinds of bounding-box collision detection methods. The first one uses axis-aligned bounding-boxes (AABB). In this method each object has two 3D points that describe the AABB and to determine if two objects collide a set of "if" conditions is used to check if

the bounding-boxes intersect each other. This method is simple and fast but with the speed and simplicity comes a limitation: as the boxes must be axis-aligned, rotating objects will seriously jeopardize the collision detection's accuracy. However, a different approach solves that problem easily.

The second way to detect collisions on bounding-boxes is to use object oriented bounding-boxes (OBB). In this approach each object needs to have a set of points that describes the object's bounding-box in a default position (object was not moved scaled nor rotated) and a World Matrix of an object. To test two objects for collision first object's bounding-box is transformed by its matrix and after that it is transformed by second object's inverse matrix. After the transformation is done the second bounding-box is axis-aligned and the first one is oriented. To do a proper test for collisions all the edges of the oriented box must be tested against the axis-aligned box using a set of if conditions to check if any of its edges collide with volume of the axis-aligned box. If they do then the collision has been detected.

Using bounding-boxes in favor of bounding-spheres gives much better results and allows for a much more accurate collision detection system, which in many cases provides enough precision. But there are situations where the mesh-to-mesh collision system is necessary.

2.14.3. Mesh-to-Mesh Collisions

Mesh-to-mesh collision system is the most precise and accurate of all mentioned in this chapter. However, the accuracy comes at a high cost, as this method is the most time and resource consuming. That is why it is only used after detecting collisions using bounding-spheres or bounding-boxes. The collision test is achieved by testing all edges from first mesh against the second mesh and if no collision is detected all edges from second mesh are tested against the first one. Testing an edge against a model is achieved by using the intersect function provided by the Direct3D API, which tests all triangles from the mesh against a ray.

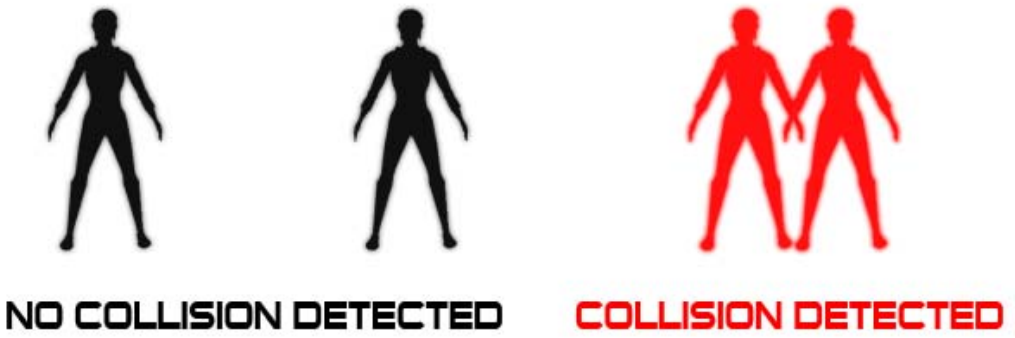

Figure 40: Mesh-to-Mesh collision detection

2.15. Artificial Intelligence

In real-life all lifeforms have certain behaviors that are common to their species. They react to different things in different ways depending on the situation. In games, the player often encounters humans or animals and expects them to behave in a realistic way. For example, some animals may be expected to be very hostile and dangerous, while other should just run away. The same is true for humans. Depending on their affiliation or the situation in the game they should also react to the player and act accordingly. Such behaviors are simulated by a set of more or less complex algorithms, which are collectively referred to as the artificial intelligence.

2.15.1. Artificial Intelligence in Games

Artificial intelligence is one of the most complex and interesting fields of research and is used in computer games extensively. By simulating more complex human, animal or even alien behaviors, designers and programmers are able to create a more interesting and challenging gameplay. Some games do not require any artificial intelligence systems at all, as their gameplay is concentrated on activities such as exploration or puzzle-solving. In action-oriented games, however, it is very much needed.

Artificial intelligence is used to control the game's units (human, animals, vehicles, etc.). This can, however, include a wide range of different actions depending on the style of the game. For example, in RTS games, artificial intelligence system would control whole groups of units depending on the actions of the player and the strategy the artificial intelligence algorithm decided on, while in action games it might only move the enemies according to the predefined patrolling paths. Especially in RTS games, but also sometimes in cRPGs, the artificial intelligence system learns from the actions of the player. If the player performs the same kinds of action repeatedly, the enemy can try to adapt its strategy to defend itself better and as a result make player's attacks unsuccessful. Additionally, it can observe the weak points of the player and try to attack those instead.

Creation of a good artificial intelligence system is a very complicated and time consuming process. Even though there are many known techniques developed in academic circles they can not always be directly applied to games, mainly because of performance issues, but also for the reasons of incompatibility with different structures present in the game. It is important to remember that all chosen algorithms and solutions must be very efficient and effective, as not much time can be allowed for all the calculations to be performed between the two

consecutive rendering cycles. However, a good artificial intelligence system can not be too effective. It must protect itself from always generating the best solutions and actions [Rabin00-1].

2.15.2. Pathfinding

While simple for humans, pathfinding is a much more complex problem for computer controlled units. It needs to be simulated using a set of algorithms and environment layout information. The are many situations that need to be considered while designing the pathfinding algorithm for a game. Units should be able to avoid obstacles, move towards a goal without falling into dead-ends or going in loops and should eventually choose a route that is easy to travel and relatively short. A perfect pathfinding algorithm does not exist but with enough careful thinking, design and planning it is possible to create a very effective solution [Stout00] [Rabin00-2] [Snook00].

2.15.3. A* Algorithm

A* [Hart68] is one of the oldest[1] and at the same time the one of most commonly used algorithms for pathfinding in computer games. It uses a set of connected nodes with assigned travel costs from each node to the other and is always able to find a path from the start node to the end node if such a path actually exists. The search process uses two different lists called Open and Closed, which contain unchecked and checked nodes accordingly. At the beginning of the search the Closed list is empty, while Open contains only the start node. In each iteration the algorithm chooses the node with the smallest traveling cost from the start node. If that node is the end node then the path has been found and the algorithm ends. Otherwise, the chosen node is inserted into the Closed list and all its neighbors are inserted into the Open list, after which the algorithm repeats its operations. If the Open list becomes empty then no path from the start node to the end node exists.

As mentioned, the A* algorithm uses a set of connected nodes to perform the path search. Since the nodes can be anything it is not a surprise that the faces of a polygonal mesh can also be used as nodes. Such mesh is referred to as a **Navigation Mesh** and is a very popular and simple technique of defining areas on which computer controlled characters can walk.

2.15.4. State Machines

In real-life, humans and animals can behave in many different ways depending on the

[1] A* algorithm was first described in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael.

time, their mood or the overall situation. Their behaviors can change if a certain event takes place or enough time passes. Because of the complexity of computer games and their gameplay designs it was important to allow computer controlled units to easily change their behaviour according to certain predefined events. A way to do that was to use **finite state machines** to design the units' behaviour models [Dybsand00].

A finite state machine is a concept which uses **states**, **transitions**, **events** and **actions** to describe a complete model of behaviour of an object. A state describes the object's current behaviour and what actions it can perform while in that state. Transitions describe events that need to take place in order for the object to switch from one state to another and the actions which take place during the transition. An action is a description of an activity the object can perform and depending on the state the object is in different actions are performed. There are four kinds of actions:

- Entry Action – which is performed when the object enters a state
- Exit Action – which is performed when the object leaves a state
- State Action – which is performed while the object is in a state
- Transition Action – which is performed while switching from one state to another

This simple concept allows designers and programmers to easily map and implement the behaviour patterns of all computer controlled units. For example, if an object should patrol the area in its default state and fight only if the player attacks it then it would have the **"Patrol"** and **"Fight"** states with the transition event from one to the other being **"Attacked by the Player"**.

2.15.5. Events and Messengers

The behaviour of computer controlled units changes only if a certain event takes place. This makes it necessary to create enough functionality to make sure the object changes the state when such event takes place. An obvious solution would be to have the object check if the event took place every update cycle. To save computation time, however, a completely different approach can be used. Instead of having the object check if the event took place, it is the event that informs the object about it.

The objects are usually informed about all events by a messenger system, which handles all the messages sent by all objects. Since all messages are sent through one system it is possible to record them in a log, as they contain information about the sender, receiver, the event that took place. This makes it easy to track the messages in order to analyze and debug the game's artificial intelligence system.

2.16. Theoretical Section Wrap-up

As proven by the following section, it is necessary to research many different topics from many different fields of knowledge in order to be able to write a more or less complex real-time application in a form of a computer game. A programmer working on the game's engine must be knowledgeable in topics like vector and matrix mathematics, 3D modelling basics, light and texturing physics, collision detection, artificial intelligence theory among many others. Of course, in bigger projects there are several groups of programmers, each with specific tasks and areas of expertise. However, in the case of the "Nyx" project, the authors needed to learn and research all of those topics to be able to create first a working framework and engine, and later a working game.

3. Design & Implementation

The goal of the theoretical section was to inform the reader of all technologies and approaches used in the project this thesis describes. This section, on the other hand, will describe the exact methods used by the authors during the implementation stage based on the theoretical knowledge presented in the previous section.

The topics presented in the following section include:

- Preparation of game design and engine design
- The engine's basic architecture
- In-depth look at the renderer and effects used
- Collisions, space division, sound and particle systems as additional engine modules
- Design and implementation of gameplay and artificial intelligence
- Level editor as a tool for content and gameplay control

3.1. The Design Process

Before implementation of the "Nyx" project could begin, it had to go through a design and analysis stage, during which all the gameplay mechanics were defined and the engine's structure was drafted. This stage was perhaps the most important of all as the decisions made at that point had their effects on the whole development process right up to the very end.

During the design stage a general goal list was compiled, which helped decide what needed to be implemented in order to achieve all those goals and also allowed to create certain safeguards by carefully planning certain application architecture structures. Moreover, a general plan of action was also drafted and contained the order in which all of the application's modules needed to be implemented.

3.1.1. Game Design

Figure 41: Concept sketch of the game's main character, Sian.

As mentioned at the very beginning of the thesis, the project in question is a computer game fully designed and implemented by the authors. It began as any game project begins – with an idea. Formulating the idea for the game was not a simple task and before it was decided what the game would be like and what it would be about many ideas were discarded in favor of newer and better ones. Eventually, however, the authors decided to create a game from an action/adventure genre.

Still, the idea was not enough and several documents detailing the game's world, storyline, characters and missions had to be created. The authors created a very detailed and rich world with full-fledged characters and missions. However, they have soon realized that the task of creating a larger game in a small and inexperienced team would be an impossible task to accomplish within a timeframe of 11 months. Therefore, the design had to be limited to the basic gameplay elements and the whole storyline was discarded in favor of a smaller story and setting. Many of the gameplay elements received a secondary priority and would only be implemented if there was enough time left before the deadline.

The game would be set in a fictional world of Emmery and would be about a female ex-thief named Sian (pronounced "see-arn"). Sian would be member of a "clan of heroes", a society of individuals working together to protect the lands and inhabitants of the world from crime. The game's design would allow the player to use Sian's skills to infiltrate the game's locations and complete tasks while avoiding the guards and enemies. Staying undetected would be the main goal of the player, which would make stealth one of the primary elements of the game's design. Of course, in case the player would be detected it would be possible to fight the enemies. However, it would be very hard as the enemies would be tough to defeat.
The authors made a decision to create a game based on avoiding conflict rather than openly allowing the player to fight. Instead of creating an environment where the player would be able fight multiple enemies at once and somehow survive after receiving extreme amounts of damage, a more realistic approach would be used. In "Nyx" the player would be allowed to fight but would only be able to survive a small amount of hits before dying.

Stealth introduces a completely different type of gameplay. Simple situations, which could be solved with an overpowered character capable of fighting ten or more enemies at once and defeating them all, suddenly can become very delicate and tricky. Such situations would require an alternate, more elaborate and careful solution. This creates an additional challenge for the players and greatly enhances their involvement in the game.

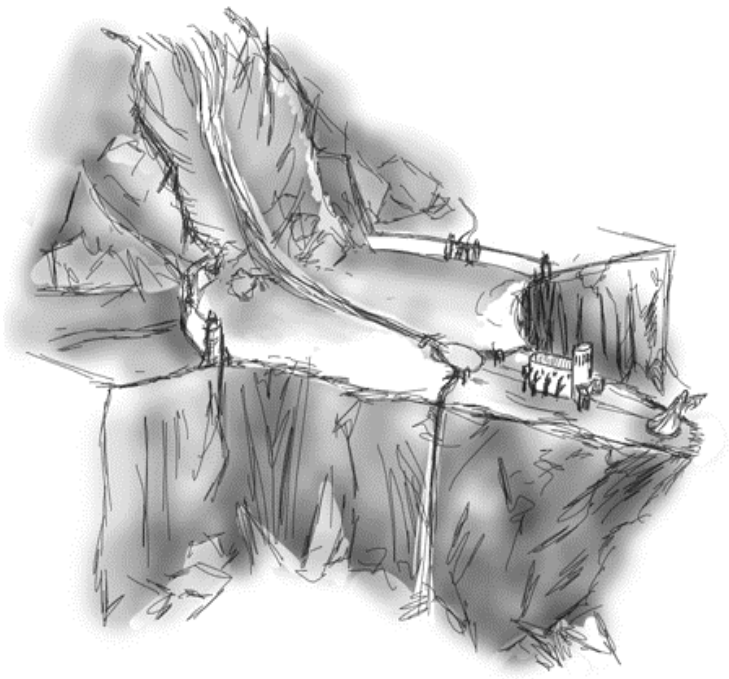

Figure 42: Concept art of the game's main location

Finishing the design did not mean the end of the design process itself. Several gameplay elements went through additional changes as time went by, especially the story and locations. Originally the plan was to have the main character being locked in a high security prison after being captured by the game's villains and the goal of the player was to free her. The whole mission would therefore take place in a confined space of an underground building. About three months into the project's implementation, the authors decided that they were not satisfied with such approach and changed the setting to an outdoor location – a city.

There were several reasons for that. The main reason was a slight change in the engine's design, which allowed for more freedom in content creation. Other reasons were purely artistic. The authors felt that a more "moody" and intense atmosphere could be created by using a city location as the basis for the game's missions, especially since the game would now be set during night time.

3.1.2. Team Assembly

Real-time rendered games are rarely created by one person and while it is possible it requires from the developer to be fluent in both programming and 3D modelling. It was obvious from the start that a great deal of graphical content would be required to complete "Nyx". The locations and characters would need to be modelled and textured and additionally all of the characters would need to be skinned and animated according to the needs of the game's design. Theoretically, all those tasks could have been done by the authors to some extent but since it was not a requirement for the authors to also work on the graphical content creation it was decided that a better choice would be to ask people with more experience in 3D modelling and texturing for help. An additional team of people whose only task would be to create all the models and textures would allow the authors to concentrate exclusively on engine's and game's design and programming. As a result it would be possible to implement many more techniques and more time would be available to polish all of the game's elements.

Eventually several 3D Artists and a Concept Artist were brought to the team to create the graphical content and help with the game's creation. Their help and expertise was invaluable and made many of the techniques presented in the project and this thesis possible.

3.1.3. Technology and Conventions

Before the implementation process could begin several issues needed to be considered, ranging from the choice of an integrated development environment (IDE) to the engine's structure design. It was necessary to make certain decisions very early as they would

influence the whole development process immensely.

3.1.3.1. IDE and Standardization

The project was implemented in C++ using Visual Studio .NET 2003 as the IDE of choice and DirectX as a set of APIs used for real-time graphics rendering, sound management and keyboard and mouse input control. Additionally Win32 API[1] was used for window creation and handling of certain window events, such as closing the window or quitting the application. The level editor created for the purposes of the project made a more extensive use of Win32 API with GUI[2] creation and event handling. All of the classes were implemented according to OOP[3] principles including object encapsulation, inheritance and polymorphism. Also used were the singleton design pattern and some of the STL[4] container classes.

An additional tool called Visual SourceSafe[5] was used to take care of project's versioning, which allowed the authors to work on the same instance of the project at the same time. Without such a tool the authors would be forced to work on separate instances of the project, which then would have to be merged into one main copy.

3.1.3.2. Management and Naming Conventions

Because of a high number of classes and more than 20,000 lines of code several management and naming conventions had to be used to increase both code readability and to provide exact purpose definitions of each class. It was important that just by looking at a part of code it would be possible to quickly deduce its general purpose in relation to the class and the whole project.

All of the classes were implemented in separate files. All header files (***.h***) contained class declarations, while all source code files (***.cpp***) contained class definitions. Furthermore, they were divided into several categories:

- **Globals** – containing globally used information such as macros, global defines and global includes

---

[1] Win32 API or Windows API is a set of application programming interfaces for the Windows operating systems and provides functionality allowing creation, control, message handling and destruction of Windows GUI objects.

[2] GUI – Graphical User Interface

[3] OOP – Object Oriented Programming

[4] STL – Standard Template Library, which is included in the C++ Standard Library. It contains implementation of containers, algorithms and iterators, which can be used on any kind of an object.

[5] Visual Source Safe - http://msdn.microsoft.com/vstudio/products/vssafe/

- **Skeleton** – containing the application's main classes (backbone classes)
- **Managers** – containing object, sound, mesh and other singleton manager classes
- **Object** – containing all game object classes
- **Meshes** – containing DirectX mesh and animated mesh object wrappers used by the game objects
- **Avatars** – containing objects, which are skinned and animated game characters
- **AI** – containing all classes strictly related to artificial intelligence concepts
- **Sound** – containing ambient and 3D sound wrappers for DirectSound objects
- **GUI** – containing classes related to the graphical user interface and heads-up-displays
- **FX Files** – containing all of the game's shader files

Thanks to this it became easier to browse the files and the general purpose and content of each class was clear.

Every variable defined in the code needs to have a name but very often, if poorly named, it is impossible to judge the general purpose of the variable or what it contains. Of course, the name could be more descriptive and for example instead of naming a variable "Speed" it could be named "SpeedOfGuardMovement". Still, it is impossible to guess what the variable contains from such a name. Is "SpeedOfGuardMovement" a float value or a 3-dimensional vector or maybe it is a pointer to one or the other? One way to convey such information would be to make the name of the variable even longer to include the type of information it contains. Another way would be to use a special prefix notation, for example a very popular hungarian notation.

The hungarian notation uses short prefixes to provide descriptions of the type and scope of the variables. Nonetheless, this method has its drawbacks as one standardized version of it does not exist. In truth, almost every single programmer uses their own version of the notation. Every method is based on similar concepts but some prefixes always wary depending on the programmer's point of view and preferences.

The authors realized that to help improve communication between them and make the code more clear a hungarian notation standard had to be defined. The following prefixes were agreed upon:

Based on variable's membership:

- **m_** – prefix defining class attributes
- **g_** – prefix defining global variables

- **no prefix** for temporary variables

Based on variable type:

- **i** – prefix defining an **integer** primitive
- **u** – prefix defining an **unsigned integer** primitive
- **b** – prefix defining a **boolean**
- **str** – prefix defining a **string** (**STL string** or **c-strings (const char*)**)
- **f** – prefix defining a **float** primitive
- **d** – prefix defining a **double** primitive
- **dw** – prefix defining a **DWORD (32-bit value)**
- **h** – prefix defining a **handle** (especially used with Win32 API)
- **v** – prefix defining a vector (**D3DXVECTOR2, D3DXVECTOR3, D3DXVECTOR4**, which are present in Direct3D as structures containing two, three or four floats and operator overloads allowing vector operations to be easily performed)
- **mat** – prefix defining a Direct3D 4x4 matrix structure (**D3DXMATRIX**)
- **p** – prefix defining a pointer
- **pp** – prefix defining a double pointer
- **vec** – prefix defining a template **STL vector** object
- **map** – prefix defining a template **STL map** object
- **list** – prefix defining a template **STL list** object
- **iter** – prefix defining an **STL iterator**

Since prefixes alone can not provide any additional information except the scope and type, all of the variables had to be carefully named to provide clear description of their purpose and data contained within them.

Additional naming conventions included rules that all methods had to be named with the first letter being a capital letter, all defines being named using only capital letters and that classes had to be named with the first letter being a capital letter and an additional prefix defining the type of the class:

- **C** – for a standard class
- **A** – for an abstract class
- **I** – for an interface class

3.1.4. Engine Design

It is important to carefully plan the application's structure before beginning its implementation. The amount of possible planning and design decisions depends greatly on the experience of the developers. However, even with small amount of experience of working with very complex applications, it is important to try to anticipate as many issues and problems as possible and try to come with ways to solve them in advance.

The authors were faced with several different issues that needed to be solved before the implementation of "Nyx" could begin. One of them dealt with the fact that the whole application had to be composed out of different, seemingly not connected modules. Rendering, Artificial Intelligence, Sound, Animations, Collisions – all those and several other parts needed to be able to freely communicate with each other to perform their tasks efficiently. It was important to create a sort-of a map of the application where every class or set of classes would be placed and required connections could be drawn. The Unified Modelling Language (UML) allowed to design all the classes and map out the relationships between them. It also helped design the whole artificial intelligence module as well as prepare state machines for the computer controlled characters.

The application would contain several backbone classes, which would control the lifetime of the whole application as well as all of the game's objects and other modules. All objects would be updated each frame and would perform necessary operations during the update using any additional modules they could require. All objects would have a certain structure which would be ascertained using class inheritance and polimorphism. Additionally, the rendering process would be divided into three separate stages: pre-processing, rendering and post-processing.

Having a rough draft of the engine's structure made it possible to consider other issues such as how the game's world would be represented and rendered. There were two possible approaches. One would take advantage of the tile technology and would require the artists to create of a set of building blocks of a fixed size, which later on would be put together to create highly customizable areas and locations. The idea was to have tiles of standard size (1x1x1, where one unit would be a predefined constant) and tiles which would be n-times bigger than the standard ones (for example, 2x1x3). It would then be possible to merge bigger and smaller tiles together to form complete game levels.

The other approach was to build the game's world as a set of complete object meshes, giving the artists a complete freedom over the looks of the environments. The advantage of

such method was that the game's areas would not look like they were built from the same reused blocks and would create more interesting environments. The disadvantage came from the fact that it was no longer possible to easily ascertain whether a certain area of the world was visible or not, which was simple to do with tiles. However, as eventually this was the approach decided on it eventually led to a development of an OctTree space division system, which would take care of dividing the game's world into smaller blocks and then rendering only those which are within the camera's viewing frustum.

3.1.5. Conclusions

While not presenting all of the aspects of the design stage of the project, this chapter provided the descriptions of the most basic elements that needed to be considered by the authors. The next chapters will provide more detailed descriptions of many of the engine's modules in the general order they were created and implemented.

3.2. Basic Architecture Elements

Every application can be divided into several elements. Each and every one of them performs different tasks and while some are more specialized in one specific area, other perform functions that take care of the more general tasks such as object management and application's initialization and de-initialization. The following chapter will explain several of the most basic mechanisms and object structures used in the "Nyx" project, while providing a necessary background for the next chapters.

3.2.1. Application's Backbone

Most applications are event based and perform certain operations only when triggered by the user. Real-time applications, however, are structured in a completely different manner. The best way to imagine how a real-time application can be structured is to take a look at any computer game with real-time generated graphics. Such game has to create enough renders every second so that the application would be interactive and the animations smooth. To achieve that all it has to perform a certain set of operations every render (or frame) in a loop. During each loop cycle all objects are updated, with the time that passes between two consecutive frames being taken into account, and then rendered. Depending on the amount of work performed, each such cycle can last from several milliseconds to even several hundred milliseconds or more.

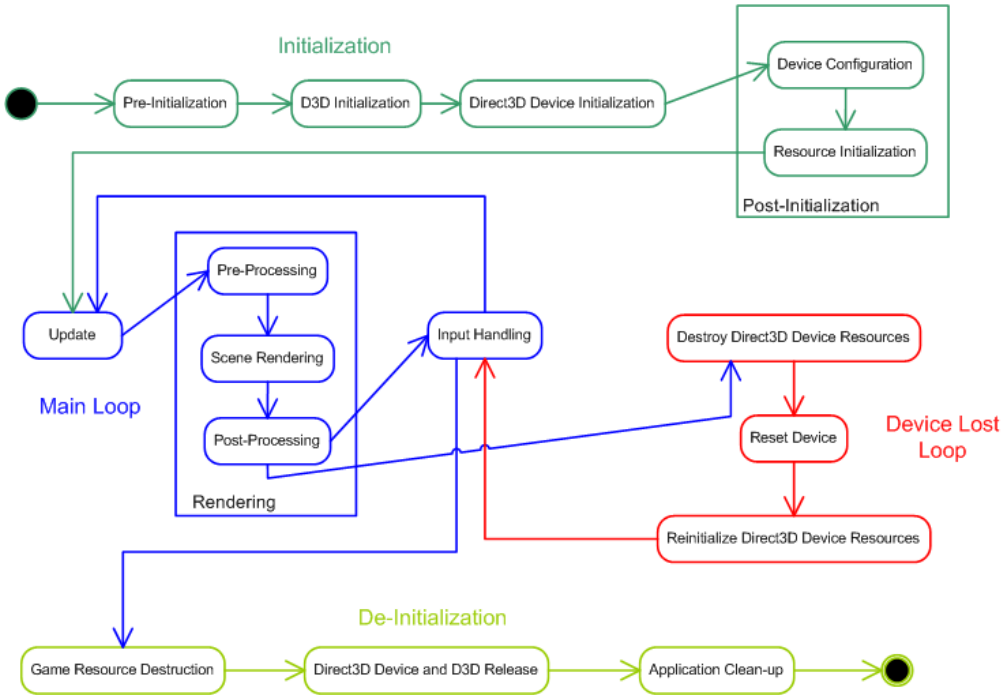

Figure 43: The diagram of the "Nyx" application's backbone.

The rendering loop is just one part of the backbone of a real-time application's structure. The "Nyx" project's backbone was divided into four different stages: initialization, main loop, device lost loop and de-initialization. Each of the four parts performed different operations but each and every one was very important. The diagram seen in *Figure 43* presents the way the "Nyx" backbone was structured and demonstrates the order of performed operations.

3.2.1.1. Initialization Stage

The initialization stage of the "Nyx" application's backbone was divided into four different parts: **Pre-Initialization**, **D3D Initialization**, **Direct3D Device Initialization** and **Post-Initialization**.

Pre-Initialization takes care of creating and registering the game's window and preloading data files containing information required to initialize the Direct3D object and device. The loaded data files contain information such as screen resolution, render quality settings, antialiasing and filtering options.

D3D Initialization handles the creation of the Direct3D interface between the application and the Direct3D API. The interface is used to create and enumerate Direct3D devices and check their capabilities as well as enumerate screen adapters and their display modes.

Direct3D Device Initialization uses the previously created Direct3D object to create an interface between the application and the graphics accelerator. Before this stage finishes with the creation of the Direct3D device it prepares a set of presentation and creation parameters,

which include the information preloaded during the pre-initialization stage as well as information such as device type and vertex processing mode. If a hardware abstraction layer[1] for the graphics accelerator is present than the device is created as a HAL device and can support both hardware and software vertex processing. Otherwise, the Direct3D device is created as a reference rasterizer[2] (REF device) and can only support software vertex processing.

Post-Initialization stage is divided into two sub-stages. The first configures the device, setting the required rendering states, such as amount of antialiasing, type of mesh face culling or alpha blending among many others, as well as all texture sampler states, such as filtering types. The second sub-stage loads all resources that are required by the application at start time. Those can include meshes, textures, shaders, game objects, sounds, etc. The resources loaded at such stage usually are global resources and are used very often throughout the application's lifetime.

3.2.1.2. Main Loop Stage

The main loop is the main part of the application. Inside it, all objects are updated and rendered and the application's interactivity is handled. Similarly to the previous one, this stage has been divided into several sub-stages: **Update**, **Rendering** and **Input Handling**.

Update, which is the first of three, performs two different tasks. First and foremost, the amount of time elapsed since the previous update is calculated based on the current system time and the time of the previous update. This is very important as all objects, and especially movements and animations, are updated with the update time being the most important factor. The reason is that every main loop cycle can take different amounts of time to be completed so moving objects by a fixed distance could create jumpy sequences and would cause the objects to move a lot faster on faster computers. That is why movements and animations are calculated by multiplying the elapsed update time by the amount of movement or animation time required to be performed per second.

---

[1] Hardware Abstraction Layer (HAL) is a set of programs, which allow communication between the physical hardware and the implemented software running on the computer. Its purpose is to provide the same set of interfaces for every supported type of hardware, hiding their differences from the operating system. That way the same code can run on systems with different hardware on them.

[2] The reference rasterizer (REF) supports every feature of the Direct3D API, but those features are only supported in software. As a result the rendering on the REF device is very slow.

After the elapsed time is calculated, all present game objects that are updated. Each object uses the time provided by the application's backbone to perform all calculations and transformations it deems necessary and as a result different objects can perform completely different update operations. For example, the player's avatar will be updated depending on the moves requested by the player, while enemy objects will perform actions depending on the decisions made by their artificial intelligence algorithms.

During the **rendering** sub-stage all game objects are visualized on the screen via the programmable shader pipeline. It is a complex process, which is usually done in several different stages and will be explained in detail in consecutive chapters.

The third and final part of the main loop is **input handling**. It performs two different functions. First, all messages sent by the game's window (created using WIN32 API) are read and reacted to, and then the player's input is handled. "Nyx" uses DirectInput to read the keyboard and mouse inputs and depending on the player's actions it performs different operations accordingly.

3.2.1.3. Device Lost Stage

At the time of the "Nyx" project's implementation Direct3D devices could have two different states: operational and lost. The operational state defines a device, which is capable of performing all operations with correct results. However, due to certain events such as game window's loss of focus while in full screen mode (for example by ALT+TAB-ing out of the application) the device can be lost. During this state all device's functions will fail silently, meaning they will not return an error while not performing their operations correctly.

When the device is lost the application needs to reinitialize all un-managed resources that use the Direct3D device. What it means is that all Direct3D resources such as meshes or textures or shaders have to be released and later recreated when the device is operational again. All other objects, especially game objects, do not need to be deleted as they are not linked to the Direct3D device.

"Nyx" checks the state of the device after finishing all the rendering operations. By calling a **Present()** method on the Direct3D device it is able to check if the device is still operational (**D3D_OK** code is returned) or lost (**D3DERR_DEVICELOST** code is returned). If the device is detected to be lost then the application switches to the device lost stage, which performs three different actions. First all un-managed Direct3D resources are released. After that the application enters a loop, which checks every 100 milliseconds if it is possible to bring the device back to the operational state. Usually, the application needs to regain focus in

order for the device to be responsive again and when that happens the device needs to be reset. Should the operation be successful the application leaves the device lost loop and reinitializes all previously released Direct3D resources and returns to the main loop.

3.2.1.4. De-initialization Stage

When the application closes, many resources need to be freed. De-initialization stage performs those operations in three separate stages. First, all game resources are freed, which requires all game objects and any other objects created during the post-initialization stage to be destroyed while releasing all Direct3D resources such as meshes, textures, shaders, sounds and fonts at the same time. Then the Direct3D device and Direct3D interface are released and freed from memory, which permanently closes the application's connection between the Direct3D API and GPU. Finally the applications window is destroyed along with any resources that might still be present in memory.

3.2.2. Game Object Structure

In object-oriented programming the term object refers to an instance of any given class. However, in game programming it can have an additional and completely different meaning. A game object is a structure that can be modified and visualized in real-time. In the "Nyx" project this term refers to objects such as characters, buildings and terrains but can also refer to particles, sprites and fonts. Basically, it all depends on the way the engine is designed and the required functionality it should provide.

As explained in the theoretical section every mesh that can be rendered must first be transformed from object space to screen space, which requires the use of the world, view and projection matrices. During the design phase of the "Nyx" project it has been decided that all game objects must contain pointers to their respective mesh structures as well as vectors and matrices, which could be used to build their world matrices. Additionally, all game objects would need to provide methods that would make it possible to correctly initialize, render, update and destroy them. However, the view and projection matrices would not be stored in those objects as they would be globally common any given scene and would be sent directly to the shaders every frame from the main loop.

Such design approach required making sure that common functionality would be identical for every game object. Object-oriented programming provides very useful mechanisms such as inheritance and polymorphism that can help achieve that. Using both concepts it is possible to move common functionality of two or more classes to another class called a base class.

Other classes can inherit it and, unless specified differently by the programmer, gain access to its full functionality. This approach is very useful when two or more classes are based on an identical structure but differ in additional attributes and/or methods or even just in the implementation of the same methods.

In the case of game objects of different classes that sometimes need to be stored in the same collections or can be used by the same methods, defining a base class can greatly simplify the application's design. Not only does it allow for easier code management, modification and expansion of common functionality, but also makes operating on objects that share the same base class much easier. For example, instead of creating different collections for animated objects and normal objects, all those can be stored in one collection that stores objects inheriting the base class. The same goes for methods. Instead of creating different overloaded methods for animated objects and normal objects one method operating on the objects inheriting the base class can be created.

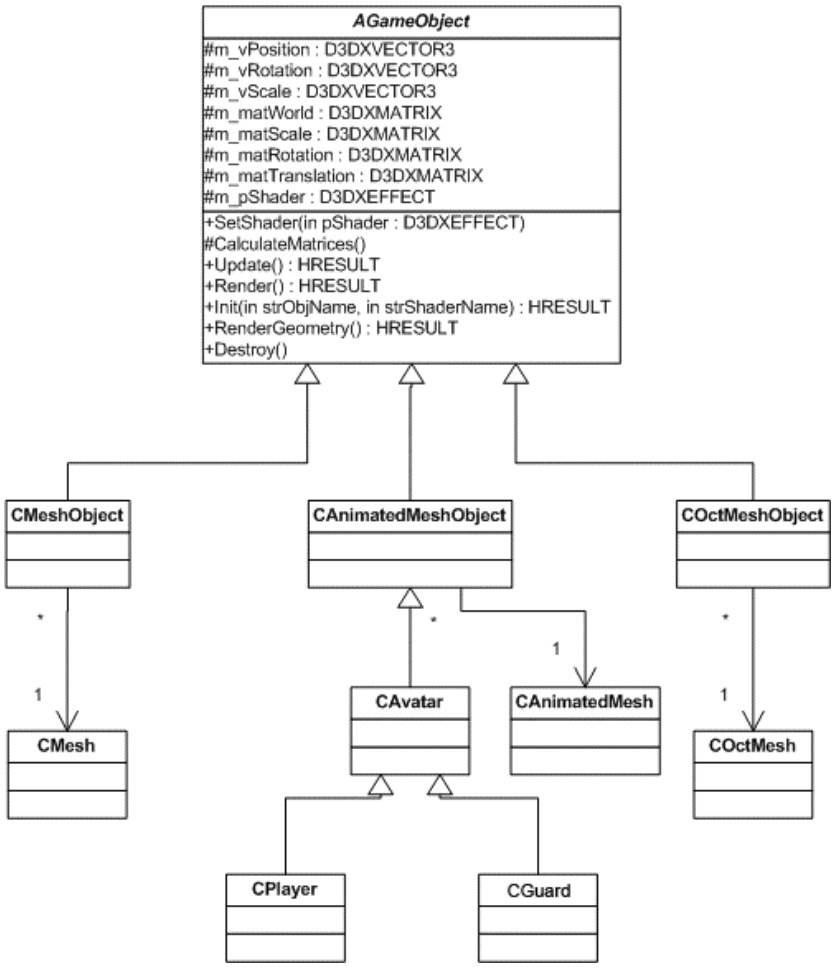

Figure 44: Diagram of the "Nyx" project's game object structure

As demonstrated in *Figure 44*, the "Nyx" project's game objects have been designed to use inheritance and polymorphism extensively. A base class **(AGameObject)** was created to provide functionality common to all game objects. It allowed easy access to all vectors and matrices related to the world matrix as well as several abstract methods, which could later be implemented by the derived classes. Additional methods and attributes included pointers to shaders and a method that calculated the world matrix.

The **AGameObject** class was designed to be an abstract class, which meant that not a single object of such class could be created. All derived classes, however, provided enough additional functionality for the class objects to be created and to be fully renderable and updateable by the engine. The project's game objects eventually were divided into three different types: mesh objects (**CMeshObject**), animated mesh objects (**CAnimatedMeshObject**) and octal tree mesh container objects (**COctMeshObject**). All of them differed greatly in their implementation but once again they followed a certain design template. Because it could be possible to render several copies of the same mesh the DirectX structures for meshes and animated meshes needed to be stored in wrapper classes. Such classes (**CMesh**, **CAnimatedMesh** and **COctMesh**) made it possible for several game objects to use the same mesh and render it in several copies keeping different attributes, such as positions or states, different for each and every instance. All **CMeshObjects** contained pointers to **CMeshes**, all **CAnimatedMeshObjects** contained pointers to **CAnimatedMeshes** and all **COctMeshObjects** contained pointers to **COctMeshes**. Such approach was especially useful for AI controlled enemies. Using just one mesh it would be possible to differently animate each and every enemy instance, locate them in different places and make them perform completely different actions.

Of course, each of those classes could have been further inherited by other derived classes. Such was the case with the **CAnimatedMeshObject** class which was further specialized to become a template for all of the game's avatar objects (**CAvatar**) and then further specialized to provide functionality specific to different types of avatars (**CPlayer** and **CGuard**).

3.2.3. Object Managers

The complex nature of real-time applications and the big amount of different data structures, requires all game objects and well as resources to be managed in an efficient and safe way. As mentioned previously, all game objects contain pointers to Direct3D mesh wrapper classes like **CMesh** or **CAnimatedMesh**. Since many game objects can point to the same mesh it is necessary to ensure that not a single mesh is loaded twice to memory.

Similarly several meshes can use the same textures, which again cannot be loaded twice. Additionally, different game objects and shaders can also be shared. This issue made it necessary to create several central resource repositories, which would allow quick and easy access to all game resources and objects and would take care of managing their initializations, updates, rendering and destruction.

All such repositories, henceforth referred to as **managers**, needed to be easily accessible from every part of the application and additionally have only one single instance. Of course it would be possible to initialize them globally, but such operation would not ensure that only a single instance could be created. Instead, a singleton design pattern[1] has been applied to all the managers.

The "Nyx" project required creation of six different resource managers. Each one took care of a different data structure but each followed very similar rules and design patterns.

The following is a list of all managers and their basic functionalities:

- **CFontManager** – manages all font resources while not making them available to any outside classes.
  - Initializes and destroys all font resources
  - Renders the requested text
- **CSoundManager** – manages all sound resources and makes them available to outside classes.
  - Provides loader methods, which initialize all requested sound objects of each implemented type (**CAmbientSound**, **C3DSound** and **C3DSoundPack**)
  - Provides methods allowing global control of several sound factors such as volume, listener position and sound attenuation
  - Provides methods returning requested sound objects
  - Destroys all sound resources
- **CShaderManager** – manages all shader resources while keeping all pointers to the DirectX shaders objects within the manager but also making them available to outside classes.
  - Initializes requested shader resources and returns them to the requesting

[1] A singleton design pattern ensures that a class can have only one single instance. It is done by making constructors and destructors private and providing a static method, which returns the only instance of the manager class when called.

objects
    - If a requested shader has already been loaded, returns the stored pointer to the shader object
    - Provides methods, which allow sending data to all existing shaders
    - Destroys all shader resources
- **CTextureManager** – manages all texture resources while keeping all pointers to loaded DirectX texture objects within the manager but also making them available to outside classes.
    - Initializes requested texture resources and returns them to the requesting objects
    - If a requested texture has already been loaded it is not loaded again but a stored pointer is returned
    - Provides methods, which initialize and return new render targets and empty textures and manages them
    - Destroys all texture resources
- **CMeshManager** – manages all mesh resources by keeping pointers to all loaded DirectX mesh objects within the manager and also making them available to outside classes.
    - Provides separate methods for initialization of each of the mesh resources (**CMesh**, **CAnimatedMesh** and **COctMesh**). The requested resource is initialized and returned to the requesting object. Additionally, all mesh wrapper classes perform additional initializations by loading DirectX mesh structures and using the texture manager to load all texture resources used by the meshes.
    - Destroys all mesh resources
- **CObjectManager** – manages all game objects by storing them within the manager, which are also available to outside classes
    - Initializes all possible game object structures and returns them to the requesting object
    - Provides methods, which allow to map any object to a specified name and later retrieve it using that name
    - Allows the engine to set global pre-processing shaders on all objects in the object manager
    - Destroys all game objects

3.2.4. Conclusions

Real-time applications, and especially games, require many different structures to be implemented in order to efficiently control the applications flow and lifetime. It is important to carefully build such structures as they will manage all other objects and events and more or less will define the application's basic functionality. Careful design will allow more flexibility and will allow extending the application with additional modules. The next chapters will describe several of such modules and will provide more detailed descriptions of some parts already described.

3.3. Visuals and Rendering

Computer games are known to be pushing the limits of real-time generated graphics. By using a wide range of effects and techniques, programmers and artists are able to create very realistic and astounding scenes. That is why good rendering systems are becoming more and more important in any game engine. Such systems, called renderers, are one of the more complex structures and require very specific approached to be implemented.

A renderer is a part of the engine, which visualizes all game objects in a certain order using a predefined set of rules. Just like any other part of the application the way the renderer works depends greatly on the application design decisions and especially on the functionality it must provide.

By design, the "Nyx" project was to be able to generate high-quality real-time renders and present many different rendering techniques and effects. That is why its renderer soon became one of the most complex and diversified parts of the engine.

This chapter will present the functionality of the "Nyx" project's renderer and the practical uses of some of the techniques presented in the theoretical section of this thesis.

3.3.1. General Requirements

Before work on the renderer could begin it was important to decide what effects had to be implemented. From the start it was the goal of the authors to create a renderer capable of performing Phong-based per-pixel lighting with extensive use of normal mapping. Additionally, for brick-like surfaces parallax mapping would be used to increase the quality of the renders. Those several effects created a basic list of requirements. However, because "Nyx" was supposed to take place at night, that list had to be slightly expanded. First of all, ambient and directional light sources had to be defined in such a way that they would be able to simulate night-time. Using very weak light sources would cause the objects to be very dark

and as a result they would be almost invisible on the screen. A better solution was to use a certain movie trick, which filters out red and green colors almost completely and boosts the blue colors. This way a good looking night could be created very fast. Additionally, sky and fog effects had to be generated to further increase the realism of the scenes. Fog was especially important as it allowed to gradually fade distant objects to the color of the sky or the back buffer. That way when objects would be cut off by the far plane of the viewing frustum it would not be that much noticeable. Another technique the authors wanted to implement were real-time shadows. However, to do that a small bit of realism had to be sacrificed as shadows are almost non-existent at night in real-life, unless of course artificial light sources, such as lamps of flashlights, are used to light the objects.

In addition to all the techniques mentioned so far, the authors decided to implement several post-processing effects, which would demonstrate image-space transformations.

3.3.2. The Renderer

The application backbone diagram places the "Nyx" renderer in the main loop right after all objects are updated. The rendering process is performed in three different stages:

- **Pre-processing**, which is generally used to prepare additional information needed in the later stages of the rendering process and can be used to prepare shadow maps, reflection textures for mirrors or water surfaces or to perform visibility and occlusion tests.
- **Scene Rendering**, where all objects are rendered from the correct camera view with final textures and in final locations. This stage is often performed in several stages, especially on graphics accelerators with Shader Models lower than 3.0.
- **Post-processing**, during which additional effects are applied to already rendered scenes. Those can include merging of two different scene renders or filtering the resulting image to achieve a better effect. Post-processing is always done in image-space on already rendered scenes.

3.3.3. Pre-Processing

In the "Nyx" project, the pre-processing stage is used to generate shadow maps. As explained in the theoretical section this operation requires the objects that are supposed to cast shadows to be rendered from the light source's view perspective, while saving only depth values in the resulting render. The orthogonal perspective is used to generate the shadow maps for the directional light, which lights the whole scene.

Because shadow mapping is implemented in "Nyx" using the standard approach without any perspective corrections, the resulting shadows contain very aliased edges. To slightly increase the quality of the shadows a small mechanism was implemented. The used approach is common in shader implementations for both Shader Model 2.0 and Shader Model 3.0 and generates two separate shadow maps. The first map covers a very small area and is generated only for the player's avatar and any other avatars that are within the covered area. The second map, on the other hand, is generated for a much larger area and includes depth values of the whole environment including all avatars not rendered into the previous map. Using this approach the shadows of the player's avatar and any other avatars that are close enough are very detailed, while the rest of the environment receives shadows of medium quality. Additionally, the environment shadow map is rendered in such a way that it covers only the areas visible by the player. It is done by offsetting the Position and LookAt vectors of the light's view in the direction of the player camera's LookAt vector.

The pre-processing stage is used to perform one more operation. Since the game's static environment is divided into blocks using the Octal Tree algorithm, every time it is to be rendered all blocks have to be tested if they are within the camera's viewing frustum. This ensures that only visible blocks are rendered.

3.3.4. Scene Rendering

During the second stage of the rendering process the game's scene is rendered from the game camera's perspective with all objects placed in the same positions as during pre-processing. All textures are placed on the objects and all effects are applied on the visible surfaced.

3.3.4.1. Sky-dome rendering

The first object that is rendered and textured is a simple sky-dome. Basically, in real-time generated scenes the sky can be rendered in two different ways. Either using a box or a low-poly sphere. While a box is used mainly for static sky rendering, a more complex and higher quality effects can only be achieved using a sky-dome.

The rendering theory behind sky-domes and sky-spheres requires that the center of the box or sphere is exactly at the location of the player. Additionally, as no z-buffer writes are performed, the sky is rendered as if infinitely far away from the player. Thanks to this the sky not only remains in place and creates an illusion of being very far away from the player but also does not have to be scaled at all and can be even smaller than the environment mesh.

The first sky in "Nyx" was generated using a sky-box. Six textures were created and placed on each face of the cube and created an illusion of a seamless sky. This technique required the use of very big textures (2048x2048) to achieve good looking results. After a bit of experimentation the old sky-box was replaced by a sky-dome and instead of using 6 high resolution textures only 4 lower resolution (1024x1024 and smaller) textures were used. Those textures were highly tileable and were wrapped several times on the surface of the sphere and merged with additional randomly placed starfield textures and clouds *(see Figure 45)*.

Figure 45: Comparison of two different sky generation techniques.

3.3.4.2. World Rendering

Each object is transformed and rendered in a certain order. First, the mesh divided by the Octal Tree is rendered with all additional static meshes contained within the visible nodes being rendered as well. Next, all enemy avatars are rendered with static objects attached to them and finally the player's avatar is rendered with all weapons attached to it. The reason for such an order is very simple. Because some of the weaponry carried by the enemies have areas with alpha maps applied they had to be rendered after the static world in order for the alpha blending being performed correctly. The same is true for the player's avatar. In certain situations, such as when the avatar's back is right up against a wall, the camera has to move much closer to the avatar and an additional blending needs to be performed. Basically, the closer the camera is to the avatar the more transparent the avatar has to be so that when the camera is located inside the avatar's mesh the faces are completely invisible.

3.3.4.3. Object Texturing

Every mesh needed to have a certain set of textures. The basic two that were required were the diffuse and normal maps. However, many objects required additional ones such as alpha, specular, glow or parallax maps. This required careful design and planning to ensure that all possible configurations could be handled efficiently in real-time without the need for writing different shaders for every possible way the objects could be textured. To do that and lower the amount of memory needed to store the textures the alpha textures were saved in objects' diffuse textures in the fourth (alpha) channel. Similarly the black and white specular map was saved in the fourth channel of the tangent map. The parallax and glow maps were used as a third texture where necessary. Of course, it may seem that since specular maps can be color, a better choice would be to save the black and white parallax map in the fourth channel of the tangent map and use the specular map as the third texture. However, as specular maps were more common than parallax maps and each one was black and white it was decided to store them in the normal map instead.

3.3.4.4. Object Mesh Structure

All meshes needed to be structured in a certain way. All faces had to be triangles but most importantly each of the vertices had to contain certain exact information in an exact order. Except the obvious ones like object space position and UV texture coordinates, every vertex had to contain three additional vectors: normals, binormals and tangents. Those three vectors create the face's tangent space, by defining X, Y and Z axes, which is further used to transform all view and light vectors to tangent space and as a result allowing the use of tangent-space normal maps. While binormals could be calculated from a cross product of tangents and normals, storing them in the vertex structures allowed them to be simply reused instead of forcing the vertex shaders to recalculate them for every vertex in every rendering cycle.

3.3.4.5. Object Lighting

There are only three different types of light sources present in the game. The first one is the ambient light used to uniformly light all of the objects and increase the scene's brightness. The second light source is the directional light that is used to add diffuse and specular lighting to the whole scene. The final light source is a point light, which is used to simulate torch lights carried by the enemies.

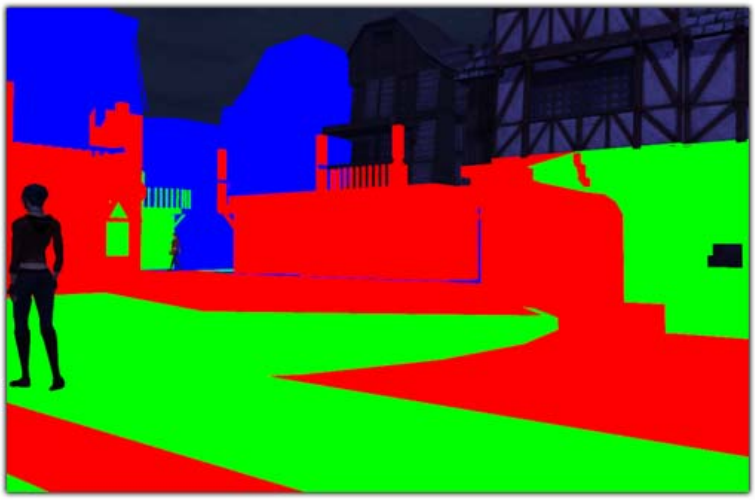

Figure 46: The visualization of the point light detection algorithm.
Blue areas are affected by one light, green by two and red by three light sources

While all objects could be rendered in one pass on Shader Model 3.0 cards, two different passes were required on Shader Model 2.0 cards. The first pass applied the directional lighting and shadowing, while the second one applied any point lights that could be affecting the rendered object. The second pass was capable of performing calculations of maximum 3 point lights per object. In any scene, however, the majority of the objects were not affected by any point lights or affected by just one of them. For that reason, a special algorithm was implemented which checked how many point lights affected any given object. If no lights were anywhere near the object then only a standard one-pass rendering was performed. If point lights were detected, however, a different shader was used in the second pass depending on the number of lights affecting the object *(see Figure 46)*.

3.3.5. Post-Processing

Because "Nyx" renders the scene to a render target texture instead of the screen buffer, additional post-processing effects can be performed by the engine, which include all three post-processing effects explained in the theoretical section. Post-processing is done in a very simple way. A small quad is used and scaled to the size of the screen buffer and the scene render is applied onto it as a texture. By manipulating the texels of such a texture it is possible to apply the additional effects.

3.3.5.1. Adaptive Glare

The bloom technique used in "Nyx" follows the ideas presented in the theoretical section, but additionally applies several modifications presented in the book ShaderX 3 [Sousa04].

The technique makes use of two different textures. The first one is the gaussian blurred glare texture calculated by squaring the color values of a downsampled scene render. The second one stores the scene's luminosity and requires the scene to be downsampled to a 2x2 texture and later averaged to a 1x1 texture. The additional information made it possible to dynamically adjust the bloom effect depending on the overall brightness of the scene. That way very bright objects will seem much brighter at night then they would during daytime.

3.3.5.2. Motion Blur

Motion blur used in "Nyx" is a very simple one. When the effect is turned on the scene render is copied to another texture and is later blended with the next frame. The result is copied to another texture, which is blended with the following frame, and so on. This way all frames that are rendered are blended into one texture. The older frames become less and less visible, while new ones create the blur effect. This approach does not produce best looking results but since the effect is only present during the player's avatar death scene a more complex motion blur was not necessary.

3.3.6. Conclusions

As shown in this chapter, the "Nyx" renderer required a lot of work and preparation of many different shader programs. Different shaders had to be prepared for different texturing and shadowing techniques and different sets of shaders were created for pre-processing and post-processing effects. However as shown, they needed to follow certain design conventions in order to be fully compatible with all meshes and texturing techniques used in the game.

3.4. Additional Engine Modules

3.4.1. Octal Tree Space Partitioning

During the game's design process it became clear that the performance of the game's renderer will be greatly affected by the complexity of the environment's geometry. It was therefore necessary to implement a module, which would cut such geometry into smaller chunks using the OctTree space partitioning algorithm and would render only those chunks that would be visible to the camera. Such module would greatly increase the speed of the rendering process and would allow more objects with more complex geometry being placed in the game's world.

The space partitioning module was implemented in a **COctNode** class. Its functionality included all the basic operations mentioned above, as well as storing and rendering of all

static meshes and choosing collision nodes for the avatars.

The class can perform the mesh division process on any given mesh. The result is a set of OctTree nodes, each of which has its bounding cube, its own mesh object and a list of its children. The division process starts by calculating the width and center of the bounding cube of the input mesh. The center of bounding cube is an arithmetic average of locations of all vertices in the mesh, while the width is calculated by finding the most distant mesh vertex from the calculated center. The next step uses this information to perform a recursive process, which divides the mesh until each and every chunk contains a number of faces equal or lower to a predefined maximal face limit per mesh. In every iteration of the process, the input mesh is divided into eight smaller meshes. For each of those meshes new bounding boxes are calculated and used to further divide the new meshes.

The second function of the module is rendering only the meshes contained in nodes visible to the camera. The process starts by calculating bounding planes for the view frustum, which can be derived from the View and Projection matrices. When the planes are calculated a test is performed which checks whether or not the root node is fully or partially included in the view figure. If the node is fully included then the process ends and the whole tree is rendered. If the node is only partially included than the visibility of its children is tested the same way recursively. Otherwise, if a node is completely outside the view frustum, it is not rendered.

The third function of the **COctNode** class is selection of nodes for collision detection. By using the avatar's position and bounding spheres of the OctTree nodes the module tests which nodes are close enough to the avatar for the collision to possibly occur.

The fourth and final function of the module is managing small static mesh objects placed in the game's world. Each node in the OctTree has a list of objects that are contained inside that node. These objects are rendered only if the node in which they are included is visible.

3.4.2. Collision Detection

The game's collision detection module was implemented as a singleton to allow all objects an easy access to its functionality. It provided a set of methods capable of detecting collisions between any two given objects by using three basic collision detection concepts: bounding-box, bounding-sphere and mesh-to-mesh collisions. All of the module's methods return a boolean value, which indicates whether or not a collision has been detected. Because there were many different objects that could collide with each other and different situations in which that could happen several collision detection methods were implemented. While sometimes a simple collision between a bounding box and a mesh was enough, some

situations required that a full mesh-to-mesh collisions had to be performed.

For avatars, a simple bounding box collision detection method was used to ensure good performance of the system. It was done in a few steps, first of which used the OctTree module to determine which nodes could possibly collide with the avatar. The second step calculated the new vertical position of the avatar, after which the avatar's bounding box was tested for collision with the geometry included in selected OctTree nodes.

Since avatars could walk on different surfaces it was important to implement a function which could play correct footstep sounds depending on the type of the surface. To do that the collision module was used to detect the texture of the environment mesh's face located directly beneath the avatar. Because every texture was bound to its corresponding surface type it was easy to detect the type of the surface on which the avatar walked and playback a correct file.

Figure 47: An avatar with rendered bone bounding boxes used during collision detection with weapons

A yet different approach had to be used to detect collisions between weapons and avatars. It required a full mesh-to-mesh collision to be implemented. If not optimized, however, this method would be very inefficient. While the meshes of the weapons contain a very small amount of faces, the meshes of the avatars contain as much as 5000 faces. Therefore it was important to create a simplified mesh of the avatar. To do that a very simple method was used. Since each avatar was skinned in 3DS Max using bipeds, every bone contained a biped mesh. Those meshes were used to calculate bounding boxes of every bones *(see Figure 47)*. When transformed with the matrices used during the animation process, those boxes would create a simplified "mesh" of the avatar.

3.4.3. Sound Manager

All sounds present in the game are managed by a singleton class, which handles their creation, playback and destruction. Similarly to all previously mentioned singleton classes, the sound manager has to be available from anywhere in the application because many different objects could require a sound being played at any given time.

There are two different sound types that are handled by the manager: ambient and 3D sounds. The difference between the two lies in the way they are played and the type of audio data they contain. Ambient sounds are stereo (contain two channels of audio data) and are always played on all channels in the same way regardless of the position of the player. 3D sounds, however, require only one channel of audio data to be contained within their files, as it is played on different channels with different intensity and volume depending on the position and orientation of the player and the position of the sound in the 3D environment.

For footstep or fight actions a set of sounds was used wrapped into a structure called a 3D Sound Pack. This structure contained slightly different sounds of the same action being performed and every time it would be played by the sound manager it would play a random sound included in the pack.

3.4.4. Particle System

The particle system is used to create different particle effects. Its roles include creation and configuration, updating and in the end destruction of all particles that are part of the generated effect. The system uses a set of parameters that are used to configure the particles in the creation process. Most of these parameters are given in ranges so the system can choose a random value from the given range when configuring a particle. Such approach guarantees that the effect created appears more random, and as a result more realistic.

During the update stage the particle system calculates how many particles should be visible (alive) and if required, creates new additional ones. Then it updates all currently visible particles using the predefined configuration parameters. If all particles are dead then the system destroys itself.

The particle system performs rendering of its particles after the whole scene is completely rendered but just before the post-processing stage begins. Because the "Nyx" particles are textured, alpha-blended quads z-buffer writes need to be turned off in order to correctly render the effect. All quads are oriented in such a way that they always face the camera. It is done by calculating an inverse ViewProjection matrix and applying the particle's position to the last column of the matrix. After all particles are rendered the z-buffer writes are enabled again.

3.5. Controls and AI: Design and Implementation

Player controls and artificial intelligence are two very factors that need to be carefully designed and implemented. During the design stage many gameplay rules have been defined that specified exactly what the players could and could not do, what weapons they could use and what actions they could perform. The similar rules had to be created for the enemies. It was necessary to decide how the enemies could behave, what they could do, how they would react to the player's presence and what they could use as weapons or tools.

This chapter concentrates on the design and implementation of player controls and artificial intelligence. Additionally, several gameplay issues will be mentioned briefly to allow a full and clear explanation of the creation process of the game's artificial intelligence.

3.5.1. Camera and Control Implementation

To handle player input, "Nyx" uses a very interesting technique called "Action Mapping". Theory behind it is that all keys are mapped to actions, each of which has a unique ID. That way several keys can easily be mapped to one action and as a result the engine has to checks only which actions are being performed by the player instead of checking every possible key event. Such approach not only simplifies the implementation process but also provides an easy way to allow the players to redefine the action keys in any way they want.

"Nyx" uses a very common implementation of Third Person Perspective character control, and for that reason its camera and control systems are very much linked with each other. The most basic rule of TPP games is that the player sees everything from a camera located at a distance from the avatar. This made it necessary for the camera to behave in a certain way. First of all the player had to be allowed to move the camera around the avatar in order to be able to better observe the surroundings. Secondly the camera needed to follow the avatar at all times. And finally under no circumstances should it go through walls or any other objects. Consequently, the camera needed to react to the surroundings in such a way that if any object was directly between the avatar and the camera's position it should move closer to the avatar.

The player can control the avatar using the standard combination of keyboard and mouse inputs. The **W**, **A**, **S**, and **D** keys define the directions the avatar can go, with **W** defining movement away from the camera, **S** defining the movement towards the camera and **A** and **D** keys defining movement to the left and right of the camera respectively. The mouse is used to rotate the camera and with the functionality of the keyboard input allows a very simple and intuitive avatar control.

3.5.2. Basic Gameplay Logic

Being the primary gameplay element, stealth helped to define most of the gameplay. Sian, the player controllable character, would need to be able to move slowly, knockout the enemies or fight them if detected. There would only be two different weapons she would be able to carry, a club and the handblades. Both weapons would allow the player to perform slightly different actions. For example, fighting enemies would only be possible with the handblades, as the club would not deal any damage at all. However, if not detected by the enemies, it would be possible to knockout or kill them with any of the weapons.

The enemies needed to be able to behave accordingly to several different possible situations and scenarios. In the default state they would patrol the surroundings and if they would detect the player in any way, they would attack them. Should they get close enough to the player, the enemies would begin to fight them with their weapon and should the player escape, the enemies would begin to search for them.

This is only a basic set of gameplay mechanics but already a set of cause-and-effect scenarios becomes apparent. Depending on various factors different avatars could have different states and perform different actions. Therefore it was important to design the artificial intelligence system in such a way that all of the characters' states and actions would be easy to implement and manage. While managing different player states was as simple as checking what weapon was used or what controls were active at the moment of the update, it was a much more complex process for the enemy states.

3.5.3. Artificial Intelligence Design and Implementation

Before any implementation could begin is was necessary to define what was really important and necessary in terms of the artificial intelligence system's functionality. Very general solutions could have been used but it would surely take more time to have a general solution perform specific actions rather than implementing specific solutions from the start. By looking at the game's designs and preparing several state charts of different actions and events it was possible to create a general list of needed functionality.

UML once again proved to be a very valuable design tool *(see Figure 48)*. Class diagrams allowed to clearly visualize the structure of the game's artificial intelligence system and state diagrams made it easy to design all the possible actions the enemies could perform.

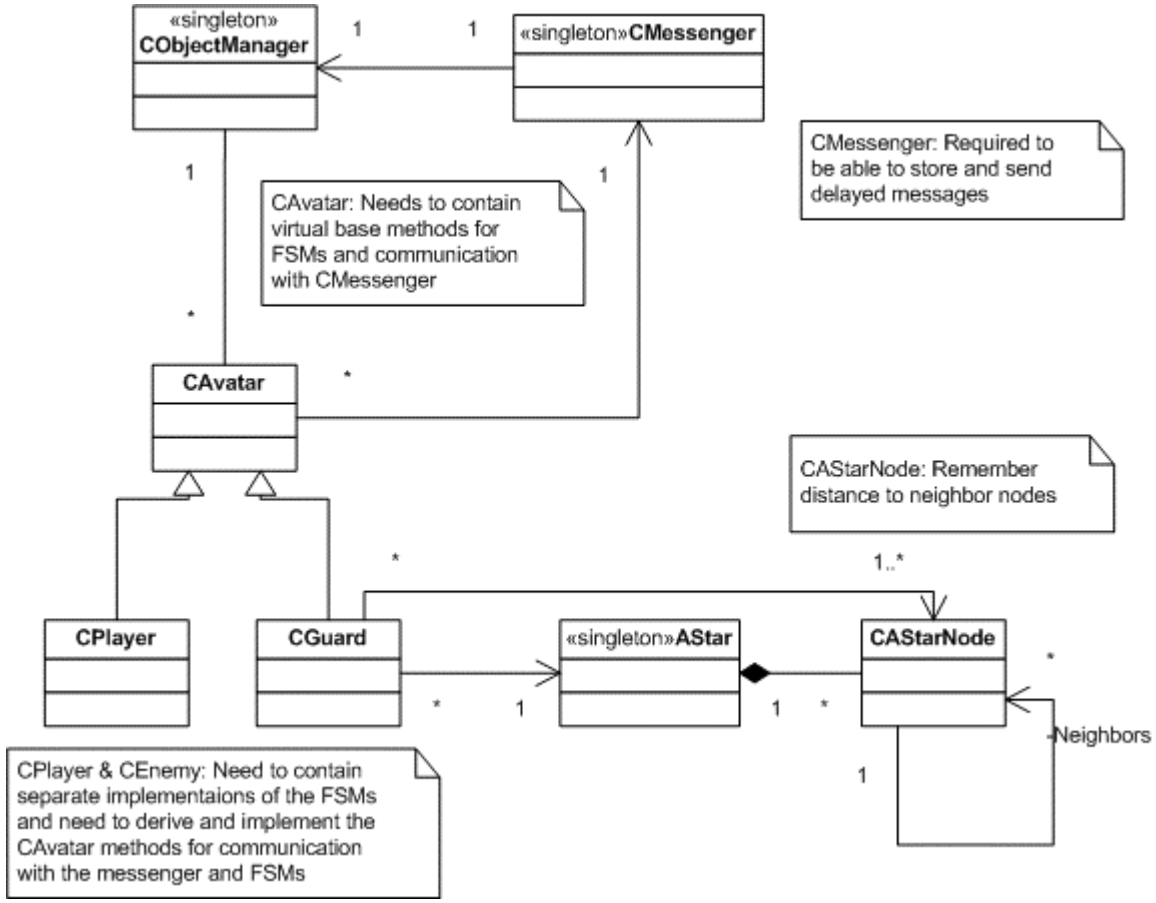

Figure 48: The class diagram of the artificial intelligence system

3.5.3.1. State Machines

State diagrams became very valuable visualizations of the ways the enemies had to behave *(see Figure 49)*. They clearly showed what actions would be performed under which conditions and what was necessary to occur in order for the enemy to switch from one state to another. But their biggest value came from the fact that they could be almost exactly translated from a diagram to the application's code. By creating three methods called **ChangeState**, **OnStateExit** and **OnStateEntry** in the avatar classes and defining a set of states as const values it was possible to implement whole state diagrams using a set of switch instruction blocks. Then during object updates the avatar's current state would be checked and corresponding actions performed. Changing the state would be only a matter of calling the **ChangeState** method after required event took place.

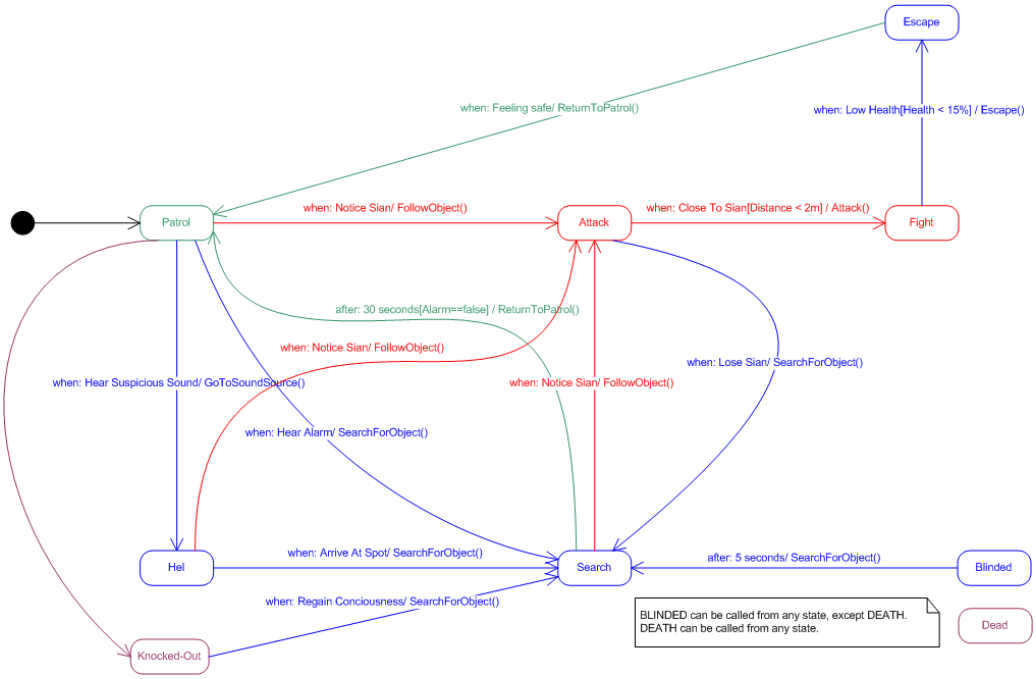

Figure 49: State diagram of CGuard class

3.5.3.2. Messengers

The **CMessenger** class allows all artificial intelligence controlled objects to send messages to each other. It contains a list of all possible events that can take place in the game. Those events can be sent at any time to any object, which simplifies communication between them. Instead of simply calling their **ChangeState** methods while performing actions, the enemies can send themselves a message, informing them about the event that took place. Then, depending on their current state and the event, the state change and additional actions can be performed. Additionally, communication via messages allows an easy way to log all state changes and events happening during the game.

As visible on the enemy state diagram some events use time values as conditions for state changes. This required an additional mechanism to be implemented in order to handle such situations. During the beginning of AI implementation, for test purposes, such conditions were checked by counting the time passed. However, by using a messenger system, such events could be handled easily and efficiently by allowing objects to send delayed messages. The messenger is able to store such messages and during each update checks if enough time passed for such messages to be sent.

Such approach allows a very simple control over timed events. For example the moment an enemy would enter the **WAIT** state a delayed message would be sent with a timestamp that would indicate the moment the message should be sent to the receiving object in order for it to return to the **PATROL** state.

3.5.3.3. Enemy Movement

Enemy movement was the final issue that needed to be considered in order to complete the design and implementation of the game's artificial intelligence system. The enemies would be completely controlled by the computer and therefore needed to be able to move in different situations in such a way that would seem natural and realistic to the player. As described in the theoretical section there are two most basic pathfinding mechanisms for artificial intelligence controlled characters in the game's world. The first is the A* algorithm and the second is a Navigational Mesh. While the first only requires a set of nodes and distances between connected nodes be known, the second would require an additional mesh. Because of the way levels were modelled and later built in the editor it was impractical to use the second technique as it would require much more additional work. Therefore it was decided to use a simple node-based A* algorithm for pathfinding. However, the game's design made it clear that not always such a solution would be necessary. The enemy state diagram allowed the following assumptions to be made:

- Patrolling state would require only patrol nodes to be placed and movement would be automatically performed from one node to the other. Any obstacles would be avoided by placing nodes in such a way that the enemy would never ever hit a wall or an obstacle.
- Attack state would require the enemy to follow the player. Instead of using the A* algorithm it would only be necessary to check if there are no obstacles between the player and the enemy and have the enemy move directly towards the player or the point where the player has last been seen.
- Should the sight of the player be lost the enemies would go to the last position in which they have seen the player and from there would use the A* algorithm to randomly walk around the level in search of the player.
- If the enemies can not find the player they would go back to patrolling and to do that A* would be used to return to the last patrol node.

Thanks to those assumptions it become clear that A* would only be used in very few situations. Therefore it was also possible to simplify the way the A* nodes would be built and connected. Instead of generating a structure similar to a walk-mesh it would only be necessary to place the nodes at corners and intersections and any other places that would seem logical for an A* node to be. Using such an intuitive approach greatly simplified the way the enemies pathfinding would be structured and produced very satisfying results.

3.5.4. Conclusions

Creating the systems of character control and artificial intelligence that would be more specific to the design requirements of the game proved to be a very good choice. Not only they required a relatively small amount of time to implement but also generated very good results, which almost fully covered the design requirements. The created system could easily be expanded further by adding additional functionality and character behaviors and actions.

3.6. NYX_LED: The Level Editor

Nowadays, development of any game requires creation of a set of tools, the purpose of which is to speed up the creation and design process of the games. Such tools can have very diverse functionalities ranging from level design to effect scripting. From the moment the development of "Nyx" began it was clear that sooner or later a level editor would have to be implemented, which would allow to design the game's levels quickly and efficiently. The only other alternatives were either hardcoding all the level information manually (which would require all objects and A* nodes being placed using trial and error method) or to use 3DS Max to place all objects and later parse such a "level" in the game's engine. In other words, there were no realistic alternatives. That is why a simple level editor was developed.

The design of the game's engine required several different functionalities to be implemented in the editor. All of those functionalities have been implemented and include:

- Setting of the environment mesh, which would later be cut by "Nyx's" OctTree algorithm
- Placement, rotation and scaling of all game objects (avatars and small meshes)
- Setting the walk-path nodes for the enemy objects
- Setting the waiting nodes and times for the enemy objects
- Placement of A* nodes
- Automatic connection of nodes that are not obstructed by any objects and calculation of the distance between them
- Saving and loading of the NYX_LED level files
- Exporting of levels to the game's level format

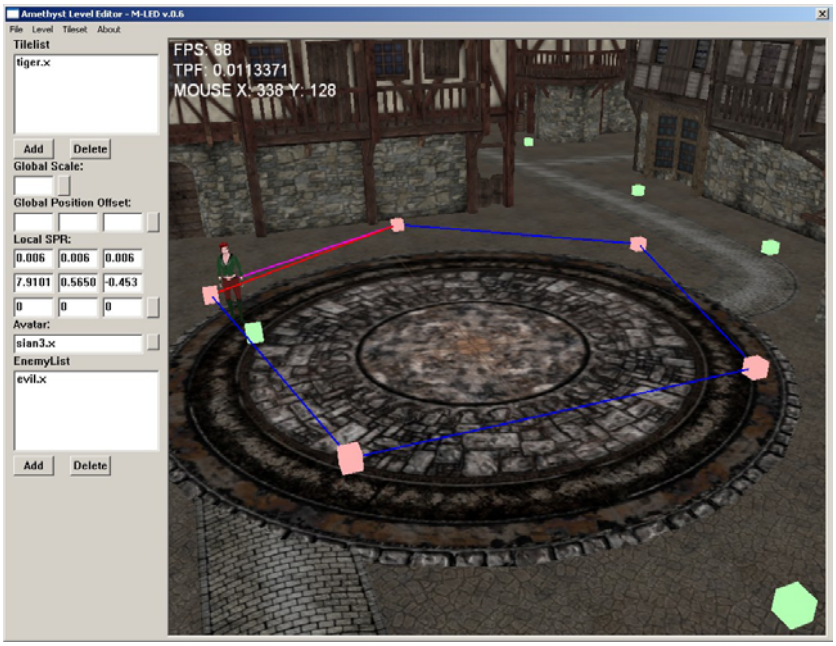

Figure 50: Screenshot from the NYX_LED level editor

NYX_LED *(see Figure 50)* turned out to be a very useful and important tool. By allowing quick level design changes to be made it helped to test new and old functionalities of the game. It was especially invaluable during the implementation of artificial intelligence algorithms and additional lighting effects added near the end of the game's implementation.

3.7. Design & Implementation Section Wrap-up

All topics covered in this section present implementation approaches of different elements of the game, which used the previously explained theoretical topics as the background for creation of a more gameplay specific systems. They illustrate that the creation of any game is a very complex process and requires a lot of planning. However, they also illustrate that with enough knowledge it is possible to easily create solutions that provide all the required functionality in an efficient way.

4. Conclusions

The following section, which is the final one of the whole thesis, contains overall conclusions and final thoughts of the authors, as well as a short overview of achieved results.

4.1. Final Results Overview

The goals of this thesis and the "Nyx" project were to demonstrate the complexity of the development process of computer games and also prove that high quality games can still be created by small teams of people. All that was done within a year's time.

While not a complete game, "Nyx" shows many different techniques and approaches used in modern games. It is not only capable of generating real-time rendered scenes of quality comparable to many of the games released in 2005 and 2006, but also utilizes many other techniques used in commercial products such as complex artificial intelligence systems, space partitioning and more.

The amount of work necessary to complete this project was enormous. It took almost a whole year to bring the project from an idea to a more or less functional game prototype. The task was especially challenging because the authors had to implement the project while learning new technologies and approaches. With more experience and knowledge it could have been completed much faster with much better results. There are many issues that could have been solved differently and in many ways more efficiently. If tasked with the creation of another game the authors would choose completely different approaches for many of the engine's areas. For example, by adding object, texture and shader sorting to the renderer it would be possible to improve its performance and by changing the structure of shaders and several managers the renderer's flexibility could be increased. However, it is important to emphasize that the authors are very satisfied with the final result as all solutions used in "Nyx" were implemented with great care and were chosen as they were considered to be both efficient and possible to implement considering the experience, knowledge and abilities of the authors at the start of the development of the project.

The amount of gameplay contained in "Nyx" is very small and the whole project should be considered as a demonstration of the basic mechanics of the game and of the engine's capabilities. It is worth noting that if additional graphical content was created the demo could be easily expanded with story and mission content as well as additional actions the player could perform.

4.2. Acknowledgements

One of the more satisfying and interesting parts of the "Nyx" project's development process was the chance of working with many different people who helped the authors complete the game. The authors would especially like to thank **Paweł Mielniczuk** for creating all of the game's 3D models and textures, **Agnieszka Zasiewska** for creating concept sketches of the game's characters and areas in the early stages of development, as well as **Krzysztof Kalinowski**, **Konstanty Kalicki**, **Filip Starzyński**, **Łukasz Wilczyński** and **Andrzej Mędrycki** for help with many of the problems the authors faced during the development of the game.

4.3. Bibliography

- [Dempski02] Kelly Dempski "Real-Time Rendering Tricks and Techniques in DirectX", Course Technology 2002
- [Blender] Anthony Zierhut's Testimonial
  http://blender.org/cms/Testimonials.261.0.html (as of December 2006)
- [Phong73] Bui Tuong Phong, *Illumination of Computer-Generated Images*, Department of Computer Science, University of Utah, UTEC-CSs-73-129, July 1973.
- [Phong75] Bui Tuong Phong, "Illumination for Computer Generated Pictures," Comm. ACM, Vol 18(6):311-317, June 1975.
- [Gouraud71] H. Gouraud, *Computer Display of Curved Surfaces*, Doctoral Thesis, University of Utah, USA, 1971.
- [Lacroix05] Jason Lacroix, "Let There Be Light!: A Unified Lighting Technique for a New Generation of Games",
  http://www.gamasutra.com/features/20050729/lacroix_01.shtml
  (as of December 2006)
- [Fernando03] Randima Fernando, Mark J. Kilgard "The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics", Addison-Wesley Professional 2003
- [Stamminger04] Marc Stamminger, George Drettakis, Carsten Dachsbacher, "Perspective Shadow Maps", "Game Programming Gems 4" Chapter 5.3, Charles River Media 2004
  http://www-sop.inria.fr/reves/Marc.Stamminger/psm/ (as of December 2006)
- [Martin04] Tobias Martin, Tiow-Seng Tan, "Anti-aliasing and Continuity with Trapezoidal Shadow Maps"
  http://www.comp.nus.edu.sg/~tants/tsm.html (as of December 2006)
- [Wimmer06] Michael Wimmer, Daniel Scherzer, "Robust Shadow Mapping with Light-Space Perspective Shadow Maps", "ShaderX 4: Advanced Rendering Techniques" Chapter 4.3, Charles River Media 2006
  http://www.cg.tuwien.ac.at/research/vr/lispsm/ (as of December 2006)
- [Futuremark06]
  http://www.futuremark.com/companyinfo/3DMark06_Whitepaper_v1_0_2.pdf

(as of December 2006)

- [DirectX1] "ShadowVolume" sample in DirectX 9.0c SDK
- [McGuire06] Morgan McGuire and Max McGuire, „Steep Parallax Mapping"
  http://graphics.cs.brown.edu/games/SteepParallax/index.html
  (as of December 2006)
- [DirectX2] "PixelMotionBlur" sample in DirectX 9.0c SDK
- [Ginsburg00] Dan Ginsburg, "Octree Construction", "Game Programming Gems 1" Chapter 4.10, Charles River Media 2000
- [Ulrich00] Thatcher Ulrich, "Loose Octrees", "Game Programming Gems 1" Chapter 4.11, Charles River Media 2000
- [Rabin00-1] Steve Rabin, "Designing a General Robust AI Engine", "Game Programming Gems 1" Chapter 3.0, Charles River Media 2000
- [Stout00] Bryan Stout, "The Basics of A* for Path Planning", "Game Programming Gems 1" Chapter 3.3, Charles River Media 2000
- [Kaiser00] Kevin Kaiser, "3D Collision Detection" , "Game Programming Gems 1" Chapter 4.5, Charles River Media 2000
- [Rabin00-2] Steve Rabin, "A* Aesthetic Optimizations", "A* Speed Optimizations", "Game Programming Gems 1" Chapters 3.4, 3.5, Charles River Media 2000
- [Snook00] Greg Snook, "Simplified 3D Movement and Pathfinding Using Navigation Meshes", "Game Programming Gems 1" Chapter 3.6, Charles River Media 2000
- [Hart68] Hart, P. E., Nilsson, N. J.; Raphael, B. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", 1968, *IEEE Transactions on Systems Science and Cybernetics SSC4* (2): pp. 100–107
- [Dybsand00] Eric Dybsand, "A Finite-State Machine Class", "Game Programming Gems 1" Chapter 3.1, Charles River Media 2000
- [Sousa04] Tiago Sousa, „Adaptive Glare", "ShaderX 3: Advanced Rendering with DirectX and OpenGL", Chapter 4.2, Wordware Publishing, Inc. 2004

4.4. Appendix A – CD Content

The CD attached to this thesis contains the following material in the following folders:

- **\DOC**
  - An electronic version of this thesis
  - Original Game Design Document
  - Original Gameplay Design Document
- **\DX**
  - Redistribution of the October 2006 version of DirectX 9.0c
- **\EXE**
  - **\NYX**
    - The game in the executable form
    - Game manual
    - **\DATA**
      - All of the game's multimedia content including music, sounds, textures and meshes exported to .X format
      - All of the game's data content, which includes: shaders, fonts and level files
  - **\NYX_LED**
    - The level editor in the executable form. Because the editor was only used as a tool to speed up the process of creation of the "Nyx" game and was not meant to be officially a part of the project it was never fully debugged and contains a number of different bugs, which did not prevent the authors from successfully designing the game's level and exporting it to the actual game.
    - A short user manual
- **\SOURCE**
  - **\NYX**
    - The full source code of the "Nyx" game
  - **\NYX_LED**
    - The full source code of the unstable version of the "Nyx" level editor

4.5. Appendix B – Game User Manual

The player controls the character using both a keyboard and a mouse. The following is a list of all controls:

- **W** – moves the character forward
- **S** – moves the character towards the camera
- **A** – moves the character to the left
- **D** – moves the character to the right
- **MOUSE** – moves the camera around the character
- **1** – using handblades
- **2** – using the club
- **F** – drawing/hiding the weapons
- **LEFT SHIFT** – running
- **LEFT CONTROL** – sneaking
- **LEFT CONTROL + LEFT SHIFT** – faster sneaking
- **E** – using smoke pellets
- **LEFT MOUSE BUTTON** – attacking with a weapon

The goal of the player is to sneak through the environment while remaining undetected by the enemies. The slower the player moves, the less chance there is of the enemies detecting the player's character. It is possible to knockout or instantly kill the enemies by sneaking behind them and attacking them with a club or the handblades. If the player's character is detected by the enemies it is possible to fight them. Another solution, however, is to use the smoke pellets to confuse the enemy for several seconds and use that time to escape and hide.

4.6. Appendix C – Index of Figures